Chapter 2. Creating and Destroying Objects（创建和销毁对象）

Item 1: Consider static factory methods instead of constructors（考虑以静态工厂方法代替构造函数）

The traditional way for a class to allow a client to obtain an instance is to provide a public constructor. There is another technique that should be a part of every programmer’s toolkit. A class can provide a public static factory method,which is simply a static method that returns an instance of the class. Here’s a simple example from Boolean (the boxed primitive class for boolean). This method translates a boolean primitive value into a Boolean object reference:

客户端获得实例的传统方式是由类提供一个公共构造函数。还有一种技术应该成为每个程序员技能树的一部分。一个类可以提供公共静态工厂方法，它只是一个返回类实例的静态方法。下面是一个来自 Boolean （boolean 的包装类）的简单示例。该方法将 boolean 基本类型转换为 Boolean 对象的引用：

```
public static Boolean valueOf(boolean b) {
        return b ? Boolean.TRUE : Boolean.FALSE;
}
```

Note that a static factory method is not the same as the Factory Method pattern from Design Patterns [Gamma95]. The static factory method described in this item has no direct equivalent in Design Patterns.

要注意的是静态工厂方法与来自设计模式的工厂方法模式不同 [Gamma95]。本条目中描述的静态工厂方法在设计模式中没有直接等价的方法。

A class can provide its clients with static factory methods instead of, or in addition to, public constructors. Providing a static factory method instead of a public constructor has both advantages and disadvantages.

除了公共构造函数，一个类还可以通过静态工厂方法提供它的客户端。使用静态工厂方法而不是公共构造函数的方式既有优点也有缺点。

One advantage of static factory methods is that, unlike constructors, they have names. If the parameters to a constructor do not, in and of themselves, describe the object being returned, a static factory with a well-chosen name is easier to use and the resulting client code easier to read. For example, the constructor BigInteger(int, int, Random), which returns a BigInteger that is probably prime, would have been better expressed as a static factory method named BigInteger.probablePrime. (This method was added in Java 4.)

静态工厂方法与构造函数相比的第一个优点，静态工厂方法有确切名称。 如果构造函数的参数本身并不能描述返回的对象，那么具有确切名称的静态工厂则更容易使用，生成的客户端代码也更容易阅读。例如，返回可能为素数的 BigInteger 类的构造函数 `BigInteger(int, int, Random)` 最好表示为名为 `BigInteger.probablePrime` 的静态工厂方法。（这个方法是在 Java 4 中添加的）

A class can have only a single constructor with a given signature. Programmers have been known to get around this restriction by providing two constructors whose parameter lists differ only in the order of their parameter types. This is a really bad idea. The user of such an API will never be able to remember which constructor is which and will end up calling the wrong one by mistake. People reading code that uses these constructors will not know what the code does without referring to the class documentation.

一个类只能有一个具有给定签名的构造函数。众所周知，程序员可以通过提供多个构造函数来绕过这个限制，这些构造函数的参数列表仅在参数类型、个数或顺序上有所不同。这真是个坏主意。面对这样一个 API，用户将永远无法记住该用哪个构造函数，并且最终会错误地调用不适合的构造函数。如果不参考类文档，阅读使用这些构造函数代码的人就不会知道代码的作用。

译注： `two` 不应是确数，应理解为概数，为绕过这个限制提供的构造可以不止两个；后半句 `...whose parameter lists differ only in the order of their parameter types.` 做了意译。

Because they have names, static factory methods don’t share the restriction discussed in the previous paragraph. In cases where a class seems to require multiple constructors with the same signature, replace the constructors with static factory methods and carefully chosen names to highlight their differences.

因为静态工厂方法有确切名称，所以它们没有前一段讨论的局限。如果一个类需要具有相同签名的多个构造函数，那么用静态工厂方法替换构造函数，并仔细选择名称以突出它们的区别。

A second advantage of static factory methods is that, unlike constructors,they are not required to create a new object each time they’re invoked. This allows immutable classes (Item 17) to use preconstructed instances, or to cache instances as they’re constructed, and dispense them repeatedly to avoid creating unnecessary duplicate objects. The Boolean.valueOf(boolean) method illustrates this technique: it never creates an object. This technique is similar to the Flyweight pattern [Gamma95]. It can greatly improve performance if equivalent objects are requested often, especially if they are expensive to create.

静态工厂方法与构造函数相比的第二个优点，静态工厂方法不需要在每次调用时创建新对象。 这允许不可变类（Item-17）使用预先构造的实例，或在构造实例时缓存实例，并重复分配它们以避免创建不必

要的重复对象。 `Boolean.valueOf(boolean)` 方法说明了这种技术：它从不创建对象。这种技术类似于享元模式 [Gamma95]。如果经常请求相同的对象，特别是在创建对象的代价很高时，它可以极大地提高性能。

The ability of static factory methods to return the same object from repeated invocations allows classes to maintain strict control over what instances exist at any time. Classes that do this are said to be instance-controlled. There are several reasons to write instance-controlled classes. Instance control allows a class to guarantee that it is a singleton (Item 3) or noninstantiable (Item 4). Also,it allows an immutable value class (Item 17) to make the guarantee that no two equal instances exist: a.equals(b) if and only if a == b. This is the basis of the Flyweight pattern [Gamma95]. Enum types (Item 34) provide this guarantee.

静态工厂方法在重复调用中能够返回相同对象，这样的能力允许类在任何时候都能严格控制存在的实例。这样的类被称为实例受控的类。编写实例受控的类有几个原因。实例控制允许一个类来保证它是一个单例（Item-3）或不可实例化的（Item-4）。同时，它允许一个不可变的值类（Item-17）保证不存在两个相同的实例： `a.equals(b)` 当且仅当 `a==b` 为 true。这是享元模式的基础 [Gamma95]。枚举类型（Item-34）提供了这种保证。

译注：原文 noninstantiable 应修改为 non-instantiable ，译为「不可实例化的」

A third advantage of static factory methods is that, unlike constructors,they can return an object of any subtype of their return type. This gives you great flexibility in choosing the class of the returned object.

静态工厂方法与构造函数相比的第三个优点，可以通过静态工厂方法获取返回类型的任何子类的对象。 这为选择返回对象的类提供了很大的灵活性。

One application of this flexibility is that an API can return objects without making their classes public. Hiding implementation classes in this fashion leads to a very compact API. This technique lends itself to interface-based frameworks (Item 20), where interfaces provide natural return types for static factory methods.

这种灵活性的一个应用是 API 可以在不公开其类的情况下返回对象。以这种方式隐藏实现类会形成一个非常紧凑的 API。这种技术适用于基于接口的框架（Item-20），其中接口为静态工厂方法提供了自然的返回类型。

Prior to Java 8, interfaces couldn't have static methods. By convention, static factory methods for an interface named Type were put in a noninstantiable companion class (Item 4) named Types. For example, the Java Collections Framework has forty-five utility implementations of its interfaces, providing unmodifiable collections, synchronized collections, and the like. Nearly all of these implementations are exported via static factory methods in one noninstantiable class (java.util.Collections). The classes of the returned objects are all nonpublic.

在 Java 8 之前，接口不能有静态方法。按照惯例，一个名为 Type 的接口的静态工厂方法被放在一个名为 Types 的不可实例化的伴随类（Item-4）中。例如，Java 的 Collections 框架有 45 个接口实用工具实现，提供了不可修改的集合、同步集合等。几乎所有这些实现都是通过一个非实例化类（`java.util.Collections`）中的静态工厂方法导出的。返回对象的类都是非公共的。

译注：原文 noninstantiable 应修改为 non-instantiable ，译为「不可实例化的」

The Collections Framework API is much smaller than it would have been had it exported forty-five separate public classes, one for each convenience implementation. It is not just the bulk of the API that is reduced but the conceptual weight: the number and difficulty of the concepts that programmers must master in order to use the API. The programmer knows that the returned object has precisely the API specified by its interface, so there is no need to read additional class documentation for the implementation class. Furthermore, using such a static factory method requires the client to refer to the returned object by interface rather than implementation class, which is generally good practice (Item 64).

Collections 框架 API 比它导出 45 个独立的公共类要小得多，每个公共类对应一个方便的实现。减少的不仅仅是 API 的数量，还有概念上的减少：程序员为了使用 API 必须掌握的概念的数量和难度。程序员知道返回的对象是由相关的接口精确地指定的，因此不需要为实现类阅读额外的类文档。此外，使用这种静态工厂方法需要客户端通过接口而不是实现类引用返回的对象，这通常是很好的做法（Item-64）。

As of（自..起） Java 8, the restriction that interfaces cannot contain static methods was eliminated, so there is typically little reason to provide a noninstantiable companion class for an interface. Many public static members that would have been at home in such a class should instead be put in the interface itself. Note,however, that it may still be necessary to put the bulk of the implementation code behind these static methods in a separate package-private class. This is because Java 8 requires all static members of an interface to be public. Java 9 allows private static methods, but static fields and static member classes are still required to be public.

自 Java 8 起，消除了接口不能包含静态方法的限制，因此通常没有理由为接口提供不可实例化的伴随类。许多公共静态成员应该放在接口本身中，而不是放在类中。但是，请注意，仍然有必要将这些静态方法背后的大部分实现代码放到单独的包私有类中。这是因为 Java 8 要求接口的所有静态成员都是公共的。Java 9 允许私有静态方法，但是静态字段和静态成员类仍然需要是公共的。

A fourth advantage of static factories is that the class of the returned object can vary from call to call as a function of the input parameters. Any subtype of the declared return type is permissible.

The class of the returned object can also vary from release to release.

静态工厂的第四个优点是，返回对象的类可以随调用的不同而变化，作为输入参数的函数。 声明的返回类型的任何子类型都是允许的。返回对象的类也可以因版本而异。

The EnumSet class (Item 36) has no public constructors, only static factories.In the OpenJDK implementation, they return an instance of one of two subclasses, depending on the size of the underlying enum type: if it has sixty-four or fewer elements, as most enum types do, the static factories return a RegularEnumSet instance, which is backed by a single long; if the enum type has sixty-five or more elements, the factories return a JumboEnumSet instance, backed by a long array.

EnumSet 类（Item-36）没有公共构造函数，只有静态工厂。在 OpenJDK 实现中，它们返回两个子类中的一个实例，这取决于底层 enum 类型的大小：如果它有 64 个或更少的元素，就像大多数 enum 类型一样，静态工厂返回一个 long 类型的 RegularEnumSet 实例；如果 enum 类型有 65 个或更多的元素，工厂将返回一个由 `long[]` 类型的 JumboEnumSet 实例。

The existence of these two implementation classes is invisible to clients. If RegularEnumSet ceased to offer performance advantages for small enum types, it could be eliminated from a future release with no ill effects. Similarly, a future release could add a third or fourth implementation of EnumSet if it proved beneficial for performance. Clients neither know nor care about the class of the object they get back from the factory; they care only that it is some subclass of EnumSet.

客户端看不到这两个实现类的存在。如果 RegularEnumSet 不再为小型 enum 类型提供性能优势，它可能会在未来的版本中被消除，而不会产生不良影响。类似地，如果事实证明 EnumSet 有益于性能，未来的版本可以添加第三或第四个 EnumSet 实现。客户端既不知道也不关心从工厂返回的对象的类；它们只关心它是 EnumSet 的某个子类。

A fifth advantage of static factories is that the class of the returned object need not exist when the class containing the method is written. Such flexible static factory methods form the basis of service provider frameworks, like the Java Database Connectivity API (JDBC). A service provider framework is a system in which providers implement a service, and the system makes the implementations available to clients, decoupling the clients from the implementations.

静态工厂的第五个优点是，当编写包含方法的类时，返回对象的类不需要存在。 这种灵活的静态工厂方法构成了服务提供者框架的基础，比如 Java 数据库连接 API（JDBC）。服务提供者框架是一个系统，其中提供者实现一个服务，系统使客户端可以使用这些实现，从而将客户端与实现分离。

There are three essential components in a service provider framework: a service interface, which represents an implementation; a provider registration API, which providers use to register implementations; and a service access API,which clients use to obtain instances of the service. The service access API may allow clients to specify criteria for choosing an implementation. In the absence of such criteria, the API returns an instance of a default implementation, or allows the client to cycle through all available implementations. The service access API is the flexible static factory that forms the basis of the service provider framework.

服务提供者框架中有三个基本组件：代表实现的服务接口；提供者注册 API，提供者使用它来注册实现，以及服务访问 API，客户端使用它来获取服务的实例。服务访问 API 允许客户端指定选择实现的标准。在没有这些条件的情况下，API 返回一个默认实现的实例，或者允许客户端循环使用所有可用的实现。服务访问 API 是灵活的静态工厂，它构成了服务提供者框架的基础。

An optional fourth component of a service provider framework is a service provider interface, which describes a factory object that produce instances of the service interface. In the absence of a service provider interface, implementations must be instantiated reflectively (Item 65). In the case of JDBC, Connection plays the part of the service interface, DriverManager.registerDriver is the provider registration API, DriverManager.getConnection is the service access API, and Driver is the service provider interface.

服务提供者框架的第四个可选组件是服务提供者接口，它描述了产生服务接口实例的工厂对象。在没有服务提供者接口的情况下，必须以反射的方式实例化实现（Item-65）。在 JDBC 中，连接扮演服务接口 DriverManager 的角色。 `DriverManager.registerDriver` 是提供商注册的 API， `DriverManager.getConnection` 是服务访问 API，驱动程序是服务提供者接口。

There are many variants of the service provider framework pattern. For example, the service access API can return a richer service interface to clients than the one furnished by providers. This is the Bridge pattern [Gamma95]. Dependency injection frameworks (Item 5) can be viewed as powerful service providers. Since Java 6, the platform includes a general-purpose service provider framework, java.util.ServiceLoader, so you needn't, and generally shouldn't, write your own (Item 59). JDBC doesn't use ServiceLoader, as the former predates the latter.

服务提供者框架模式有许多变体。例如，服务访问 API 可以向客户端返回比提供者提供的更丰富的服务接口。这是桥接模式 [Gamma95]。依赖注入框架（Item-5）可以看作是强大的服务提供者。由于是 Java 6，该平台包括一个通用服务提供者框架 `Java.util.ServiceLoader` ，所以你不需要，通常也不应该自己写（Item-59）。JDBC 不使用 ServiceLoader，因为前者比后者要早。

The main limitation of providing only static factory methods is that classes without public or protected constructors cannot be subclassed. For example, it is impossible to subclass any of the convenience implementation classes in the Collections Framework. Arguably this can be a blessing in disguise because it encourages programmers to use composition instead of inheritance (Item 18), and is required for immutable types (Item 17).

仅提供静态工厂方法的主要局限是，没有公共或受保护构造函数的类不能被子类化。 例如，不可能在集合框架中子类化任何方便的实现类。这可能是一种因祸得福的做法，因为它鼓励程序员使用组合而不是继承（Item-18），并且对于不可变的类型（Item-17）是必需的。

A second shortcoming of static factory methods is that they are hard for programmers to find. They do not stand out in API documentation in the way that constructors do, so it can be difficult to figure out how to instantiate a class that provides static factory methods instead of constructors. The Javadoc tool may someday draw attention to static factory methods. In the meantime, you can reduce this problem by drawing attention to static factories in class or interface documentation and by adhering to common naming conventions. Here are some common names for static factory methods. This list is far from exhaustive:

静态工厂方法的第二个缺点是程序员很难找到它们。 它们在 API 文档中不像构造函数那样引人注目，因此很难弄清楚如何实例化一个只提供静态工厂方法而没有构造函数的类。Javadoc 工具总有一天会关注到静态工厂方法。与此同时，你可以通过在类或接口文档中对静态工厂方法多加留意，以及遵守通用命名约定的方式来减少这个困扰。下面是一些静态工厂方法的常用名称。这个列表还远不够详尽：

- from—A type-conversion method that takes a single parameter and returns a corresponding instance of this type, for example:

from，一种型转换方法，该方法接受单个参数并返回该类型的相应实例，例如：

```
Date d = Date.from(instant);
```

- of—An aggregation method that takes multiple parameters and returns an instance of this type that incorporates them, for example:

of，一个聚合方法，它接受多个参数并返回一个包含这些参数的实例，例如：

```
Set<Rank> faceCards = EnumSet.of(JACK, QUEEN, KING);
```

- valueOf—A more verbose alternative to from and of, for example:

valueOf，一种替代 from 和 of 但更冗长的方法，例如：

```
BigInteger prime = BigInteger.valueOf(Integer.MAX_VALUE);
```

- instance or getInstance—Returns an instance that is described by its parameters (if any) but cannot be said to have the same value, for example:

instance 或 getInstance，返回一个实例，该实例由其参数（如果有的话）描述，但不具有相同的值，例如：

```
StackWalker luke = StackWalker.getInstance(options);
```

- create or newInstance—Like instance or getInstance, except that the method guarantees that each call returns a new instance, for example:

create 或 newInstance，与 instance 或 getInstance 类似，只是该方法保证每个调用都返回一个新实例，例如：

```
Object newArray = Array.newInstance(classObject, arrayLen);
```

- getType—Like getInstance, but used if the factory method is in a different class. Type is the type of object returned by the factory method, for example:

getType，类似于 getInstance，但如果工厂方法位于不同的类中，则使用此方法。其类型是工厂方法返回的对象类型，例如：

```
FileStore fs = Files.getFileStore(path);
```

- newType—Like newInstance, but used if the factory method is in a different class. Type is the type of object returned by the factory method, for example:

newType，与 newInstance 类似，但是如果工厂方法在不同的类中使用。类型是工厂方法返回的对象类型，例如：

```
BufferedReader br = Files.newBufferedReader(path);
```

- type—A concise alternative to getType and newType, for example:

type，一个用来替代 getType 和 newType 的比较简单的方式，例如：

```
List<Complaint> litany = Collections.list(legacyLitany);
```

In summary, static factory methods and public constructors both have their uses, and it pays to understand their relative merits. Often static factories are preferable, so avoid the reflex to provide public constructors without first considering static factories.

总之，静态工厂方法和公共构造器都有各自的用途，理解它们相比而言的优点是值得的。通常静态工厂的方式更可取，因此应避免在没有考虑静态工厂的情况下就提供公共构造函数。

Back to contents of the chapter（返回章节目录）

- **Next Item（下一条目）：** **Item 2: Consider a builder when faced with many constructor parameters（在面对多个构造函数参数时，请考虑构建器）**