与构造器相比，静态工厂方法的优势：

        1、静态工厂方法与构造器不同的第一大优势在于，它们有名称。使用带有名称的静态工厂方法来获得类实例可以扩展代码的可读性，更能清晰的表达程序的意图。为了实现同一个类内存在多个签名的构造器，我们往往会选择提供多个具有不同签名的构造器，但是这种方法的可读性较差，不同的构造器之间的差别仅在于参数类型的顺序上。但是如果使用静态工厂方法，就可以通过名称来清楚的描述不同的构造器需要的参数。更具可读性。

        2、静态工厂方法与构造器不同的第二大优势在于，不必在每次调用它们的时候都创建一个新的对象。这使得不可变类可以使用预先定义好的实例。避免了创建不必要的重复对象。

如果程序经常性的请求创建相同的对象，那这种技术会大大的提升性能。还有就是静态工厂能够为重复调用返回相同的对象，这样就能方便我们来控制在某个时刻都应该有哪些特定的对象实例存在，这种类被称作实例受控的类。

        实例受控的类的存在有几个原因：1、实例受控的类可以确保它是一个Singleton或者是不可实例化的。2、它还使得不可变的类可以确保不会存在两个相等的实例，即有当且仅当a==b的时候才有a.equal(b)为true。如果这得以保证，就可以使用==操作符来替代equal()方法。

        3、静态工厂方法与构造器不同的第三大优势在于，它们可以返回原返回类型的任何子类型。这种技术有一个很重要的应用就是隐藏真正的实现类，而只提供公有的基类，以这种方式隐藏实现类会使得API变得非常简洁。书中指明，这项技术适用于基于接口的框架，因为在这种框架中，接口为静态工厂方法提供了自然返回类型。接口是不能持有静态方法的，因此按惯例接口Type的静态工厂方法被放在一个名为Types的不可实例化的类中。通过这种方法，用户可以知道被返回的对象是由相关的接口定义的，所以不需要阅读有关的类文档。使用静态工厂方法时，甚至要求客户端通过接口来引用被返回的对象而不是通过实现它们的类，这是一种良好的习惯。

        静态工厂方法所返回的对象的类是可变的，这取决了传入给其的参数，其实就是一种约定的控制符。只要是已声名的实现类类型，都是可以被实例化返回的。这就大大的增加了灵活性。

与构造器相比，静态工厂方法的劣势：

1、静态工厂方法的主要缺点在于，类如果不含公有的或者受保护的构造器，就不能被子类化。这主是要由于**[Java](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "_blank)**的类继承机制的权限限制的。但是这样也会鼓励我们尽量使用类的复合而不是继承。

2、静态工厂方法的第二个缺点在于，它们与其他的静态方法实际上没有任何区别。这导致在API文档中，它们没有像构造器那样在API文档中明确的标注出来，因此对于提供了静态工厂方法而不是构造器的类来说，要想查明如何实例化一个类，这是非常困难的。可以采用惯用名的方式暂时弥补这一劣势。

     valueOf ：不太严格的讲，这个方法返回的实例与它的参数具有相同的值。实际这只是做了一个类型转换。

     of：valuefOf的一种简洁的替代，在EnumSet中使用并流行起来。

     getInstance：返回的实例是通过方法的参数来描述的，但是不能说与参数具有同样的值。参于Singleton来说，该方法就没有参数，并总是返回唯一的实现。

     newInstance：像getInstance一样，但newInstance能够确保返回的每个实例都与所有其他的实例不同。

     getType：像getInstance一样，但是在工厂方法处于不同的类型的时候使用。Type表示工厂方法所返回的对象类型。

     newType：像getInstance一样，但是在工厂方法处于不同的类型的时候使用。Type表示工厂方法所返回的对象类型。

第八条：覆盖equals时请遵守通用约定

1. 自反性。对于任何非null的引用值x,x.equals(x)必须返回true。
2. 对称性，对于任何非null的引用值x和y，当且仅当y.equals(x)返回true时，x.equals(y)必须返回true。
3. 传递性。x,y,z,如果x.equals(y)为true,y.equals(z)为true，那么x.equals(z)必须为true
4. 一致性。引用值x和y，未修改，多次调用x.equals(y)，结果必须一致
5. 对于任何非null的引用值x，x.equals(null)必须返回false.

实现高质量equals方法的诀窍：

1. 使用==操作符检查“参数是否为这个对象的引用”
2. 使用instanceof操作符检查“参数是否为正确的类型”
3. 把参数转换成正确的类型
4. 对于该类中的每个”关键”域，检查参数中的域是否与该对象中对应的域相匹配
5. 当你编写完成了equals方法之后，应该问自己三个问题：它是否是对称的、传递的、一致的
6. 覆盖equals时总要覆盖hashCode()
7. 不要企图让equals方法过于智能
8. 不要将equals声明中的Object对象替换为其他的类型

第九条：覆盖equals时总要覆盖hashCode

1. 因没有覆盖hashCode而违反的关键约定是第二条：相等的对象必须具有相等的散列码(hash code)。
2. 不要试图从散列码计算中排除掉一个对象的关键部分提高性能。虽然这样得到的散列函数运行起来可能更快，但是他的效果不见得会好，可能导致散列表慢到根本无法使用。

第十条：始终要覆盖toString

1. 在实际应用中，toString方法应该返回对象中包含的所有值得关注的信息。
2. 无论你是否决定指定格式，都应该在文档中明确地表明你的意图。

第十一条：谨慎地覆盖clone

1. 如果你覆盖了非final类中的clone方法，则应该返回一个通过调用super.clone而得到的对象。
2. 实际上，对于实现了Cloneable的类，我们总是期望它也提供一个功能适当的公有的clone()方法。
3. clone方法就是另一个构造器；你必须确保它不会伤害到原始的对象，并确保正确地创建被克隆对象中的约束条件。
4. 最好提供某些其他的途径来代替对象拷贝，或者干脆不提供这样的功能5.
5. 另一个实现对象拷贝的好方法是提供一个拷贝构造器

7. 避免使用终结方法finalize()

a) 终结方法不可预测，很危险

i. 不能保证会及时被执行，甚至不能保证执行：time-critical的任务不应该由终结方法执行。例如关闭已经打开的文件（文件描述符是很有限的资源）

ii. 不能保证在所有的JVM上都得到同样的体验

iii. 可能随意延迟其实例的回收过程

iv. 不应该依赖终结方法来更新重要的持久状态，例如分布式系统的永久锁

v. 不要使用System.gc()等函数

b) 如果在终结方法执行时抛出未捕获异常，则此异常会被忽略，终结方法也将终止

c) 非常严重的性能损失

d) 使用显式的终止方法，并要求客户端在每个实例不再有用时调用这个方法。该实例必须记录自己是否被终止，终止后的调用要抛出异常。

例子：InputStream的close函数，Timer的cancel函数等

应该在try-catch块的finally部分调用这个显式终止方法

e) 终结方法的正确使用

i. 在对象的所有者忘记调用显式终止方法时，充当安全网函数。

ii. 对象的本地对等体：终结方法释放本地对等体的重要资源

f) 子类覆盖终结方法时，需要显式调用父类的终结方法