Java API 设计指南

作者：　　Eamonn McManus
原文地址：　http://www.artima.com/weblogs/viewpost.jsp?thread=142428
译者：　　王磊
电子邮件：　wl_95421@yahoo.com.cn

前言：

市场上关于如何设计和编写优秀 Java 代码的书如此之多，可能要用汗牛充栋来形容，但是想找到一本如何设计 API 的书，却是难之又难。这里我将把自己一些关于 API 设计的经验与大家分享。

分享这些经验是源于最近我参加了 JavaPolis 上的一个讨论，这个讨论是由 Elliotte Rusty Harold 发起的，是关于设计 XOM 时的一些原则性问题，讨论中的思想交流如此精采，令我受益颇多。虽然这次讨论主题是与 XOM 有关，但是大部分的时间我们都在讨论设计 XOM API 时的一些原则性问题，而这些内容对于 API 设计而言，则是通用的。这几年，Java 的应用日益广泛，开源项目也是蒸蒸日上。一本能够指导开发人员设计和编写 API 的好书，可以帮助开发人员设计和编写更好的 API。

在过去的五年中，我一直参与 JMX　API 的修订及调整，在此过程中，同样受益颇多。特别在这次讨论会，我对 Elliotte 提出的一些观点高举双手赞同。

接下来的内容是讨论会上的一些主要观点的总结，其中包括一些个人或者是来自他人的经验，也参考一些相关的文档，希望对大家设计 API 有所裨益。

下面是个人推荐的一些阅读和参考资料。

下面给出的网址是 Netbeans 网站上的一篇关于 API 设计的优秀文档，

http://openide.netbeans.org/tutorial/api-design.html

Josh Bloch's 的 Effective Java 作为 Java 设计的圣经之一，从来都不会被漏下。

设计需要进化

API 的价值就在于能够帮助你完成许多功能，但请不要忘记要持续的改善它，否则它的价值就会逐渐减少。而且要根据用户的反馈信息，来改善 API，或者说是对 API 进行演化，另外改进 API 时要注意的就是在不同版本间要保持兼容性，这点至关重要，它也是一个 API 是否成功的重要标识之一。

如果一个功能以 API 的方式公布出来，那么在发布以后，它的对外接口就已经固定，无论什么情况，都不能取消，而且一定要能够按照原有的约定功能正确执行。如果 API 不能在版本间保持兼容性（译注：这里应该指的是向下兼容），用户将会难以接受这样一个千变万化的 API，最终的结果只能是让用户放弃使用你的 API。如果你的 API 被大量的软件或者模块所使用，又或者被大型软件使用，这个兼容问题也就愈加严重。

以一个 Java 应用程序为例，如果 Module1 使用了 Banana1.0 的 API，而 Module2 则使用了 Banana2.0 的 API，现在要同时部署这两个模块到一个 Web Application 上，如果 Banana2.0 对 Banana1.0 保持兼容的话，整个部署就会非常简单，直接使用 Banana2.0 就可以了。如果 Banana2.0 和 Banana1.0 不兼容的话，就很难同时在一个程序中同时使用 Module1 和 Module2

（可能要自定义 ClassLoader，或者是其它方式，这对于用户未免有些为难了）。最终的结果可能就是你失去一个用户，这时的 API 就不能为他人提供任何价值（译注：看来它也不能带来经济价值了）。

分析 Java SE 平台所提供的 API，有着严格的兼容性控制。其根本目标就在于保证用户在版本升级时，不会导致低版本的代码无法运行。这也再次说明，当一个 API 发布以后，任何 API 公开的方法，常量等元素都不能被移出 API，否则会严重降低 API 的价值。（译注：所以个人一向比较怀疑 deprecated 的价值，因为既然所有的方法都不会被删除，即使那些被 deprecasted 的方法和变量也会被仍然保留，其功能也不应该会被改变，因此从这个角度来说，其方法和变量的使用仍然是安全的。）

说到兼容性，必然要谈到二进制兼容性，这是指对于已经编译完成的代码，不会因为版本的变更，而致使编译后的代码不能正常运行。比如说你将一个 public 方法从 API 中移走，就会无法正常运行，会抛出 NoSuchMethodException 这类错误。

但源代码的兼容性也不可忽视，在某些特殊情况下，修改 API 时，会产生源代码兼容问题，导致源代码无法编译通过。例如添加一个重载的同名方法，其参数不同，如 getName(User)和 getName(String)，当用户使用 getName(null)时，会因为存在二义性，而产生编译错误，用户必须给出 getName((String)null)，来明确标识要调用的方法。因此必须寻找一种方式，来保持源代码的兼容性。

通常情况下，如果源代码不兼容，代码编译就无法通过，用户必须修改源代码才能保证程序正常运行，但这并不是一个好的解决方案。以 J2SE6 中的 javax.management.StandardMBean 为例，这个类的构造函数已经采用泛型了。这样会使得一些类在创建这个 Bean 的时候，会出现编译不能通过的问题，但是解决方案有时却是非常简单，往往只需要添加一个 cast 进行强类型转换就可以解决这样的一个编译问题了。但是更多的时候，需要更复杂的方案才能解决这些编译问题，例如在一个方法中调用第三方的 API 方法，如果有一天这个 API 方法的参数被修改了，在修改源代码时候，可能会发现你的方法中不含有 API 方法需要的参数，这时就需要修改方法参数。以下是这种情况的示范代码。

```
public void callApi(Parameter1 p1,Parameter2 p2)
{
    //do some operations
    new ThirdClass().doCallThirdApi(p1);
}
```

现在第三方的 API 现在变成了 doCallThirdApi(Parameter1 p1,Parameter3 p3)

这时可能需要将方法改成：

```
public void callApi(Parameter1 p1,Parameter2 p2,Parameter3 p3)
{
    //do some operations
    new ThirdClass().doCallThirdApi(p1,p3);
}
```

这样一个 API 的改变，很可能引发一个多米诺骨牌事件。

通常情况下，你不知道用户如何使用 API 来完成工作。除非能够确认 API 的修改对用户的代码不会造成破坏，才可以考虑修改 API。，反之如果 API 的改变会影响用户现在的代码，就必须有一个足够的理由。只有意识到对 API 的修改，会严重的破坏用户现有的代码，在修改 API 时才会谨慎地，尽量地保证 API 兼容性。修改 API 的时候要尽量避免以下几种情况，以免对用户的代码产生破坏：

1. 降低方法的可见性，如将 public 变成 package 或者 proteced，又或者将 protected 变成 private。
2. 修改方法的参数，如删除一个参数、添加一个参数、或者是改变参数的类型，尤以后两者更为严重。

就象业界游行的经验之谈--“不要使用 3.0 版本以下的软件”（译注：所以我总想把自己的软件直接发布为 9.9。），同样的情况对 API 也是一样的，前几个版本的 API 往往包含有大量的错误，这些错误不应该被隐藏起来，因为纵然是冰山的底部，也终有浮出水面的一天。因此在正式发布 API 1.0 以前，请不要忘记提供若干个 0.x 版本。对于使用 0.x 版本的用户，会有一个比较明确的说明，用户会清楚的知道当前版本所公布的 API 还不稳定，有可能在正式发布的时候有所更改，0.x 版本也不保证兼容性。（译注：以 Visual Studio2000 beta2 为例，与正式版的差别就非常大，所以 0.x 版本通常只是用来学习，或者进行技术预言，而不能在产品中使用）。但是一旦 1.0 版本正式发布，请记住，就是对 API 的兼容性就做出一个正式的承诺。象 JCP 组织在对某一个规范推出正式版本以前，通常都会发布若干个草稿版本（如意向草稿，公共预览草稿，最终建议版本等）。如果方便的话，对于规范性的内容，在发布 API 时，提供一个 API 的实现可能会更有效的推行规范（译注：因为规范性的内容更多的是以 Interface 的方式来发布 API，大家可以参考一下 Interface 和 Abstract Class，所以提供一个实现往往更好，象 sun 发布 J2EE 规范时，就提供了一个默认的实现。）。

当 API 的设计和开发到了一定阶段以后，可能会发现以前的版本已经出现了一些问题，又或者需要添加新的功能，此时设计人员完全可以重新创建新的 API，并放到新的包中，这样就可以保证那些使用老版本的用户可以很轻松的移植到新版本上，而不会产生问题。请牢记一点：添加新的功能，请不要修改原有的内容。

API 的设计目标

设计一个 API 要达到哪位目标呢？除了兼容性以外，也从 Elliotte 的讨论中提出一些目标。

API 的正确性必须保证：

以 XOM 为例，无论用户如何调用 API，都不应该产生错误的 XML 文档。再如 JMX，不管是注册一个错误的 MBean 还是并发执行一些操作，又或者 MBeans 使用了一些特殊的名称，MBean Server 都必须保持状态的一致性，不能在某个错误的 MBeans 进行了操作以后，整个系统就无法提供服务。

API 的易用性：

API 必须易于使用。通常易用性一向难以评价。但是有一个办法可以有效的提高易用性，就是编写大量范例代码，并将其很好的组织在一起，从而为用户提供 API 参考。（译注：个人认为一个好的 FAQ 可以提供各种 API 使用的范例。）

另外下列原则也可以用来判断 API 的易用性：

1. 是不是总是经常出现一组操作代码？（译注：这里是指如果有多行代码重复被调用，说明它们应该被放到一个方法中，避免用户重复编写一组代码。）
2. 在使用 API 时，是否需要经常参考 JavaDoc 或者是源代码，才能知道应该调用哪个方法呢？（译注：比较理想的情况就是，大部分操作只需要通过类名和方法的名称就可以明白）。
3. 根据名称调用一个方法，但是该方法所做的事并不是用户所想要的。（译注：例如

调用一个 command 方法，以为是执行一个操作，但是结果这个方法是做备份用。）

API 必须易学：

很大程度上，API 的易学和易用性是相似的，一般来说，易用也就易学。如果要使API 易学，下列基本原则要遵循的：

1. API 越小就容易学习；
2. 文档应该有范例；
3. 如果方便的话，尽可能将 API 与一些常用的 API 保持一致。例如如果要做一个资源访问的 API，尽可能与 J2SE 中的 IO 使用一致，自然很容易学习。

（译注：如果你要关闭一个资源，就象 Java 的 File,Connection 一样，使用 close，而不是 destroy。）

API 的运行速度必须够快：

Elliotte 也是考虑了很久，才给出这一条。但是要在保证 API 简单而且正确的前提下，再来考虑 API 的性能问题。在设计 API 时，你可能会先使用一种能够快速实现但是性能不好的方式来实现 API，然后再根据实际情况再修改 API 的实现，以调整性能。至于如何调整性能，绝对不要通过直觉来判断何种方式能获得高性能。只能通过正确，严格的测试以后，再对性能瓶颈进行优化从而提高性能。（译注：过早的优化是所有的错误根源，这已经是一个普遍认同的观点，特别是对于 Java 程序，因为它的 JVM 越来越快，越来越聪明。）

API 必须足够的小：

这里所说的小不仅是指编译后代码的文件比较小，而且更重要的是运行时占用的内存要小。之所以提出最小化的概念，还有一个原因：就是因为很容易为 API 添加新的内容，但是要将一个内容从 API 中移出就很困难，所以不要随便向 API 中添加内容，如果不确定一项内容，就不要将它加入到 API 中。通过这样一个建议或者说是限制，可以提醒一个 API 设计人员更加关注 API 中最重要的功能，而非一些枝节的问题。

（译注：许多时候这个最小化原则是很难遵守的，如不变类通常比可变类更好一些，但是它会占用更多的内存，而可变类占用的内存会少些，但要处理线程，并发等问题，所以更多时候是一个权衡，大固然不好，小也必就好）。

有一种设计 API 的方法很常见，但是结果却令人头痛，这种方法就是在设计 API 前，会详细的考虑每一个用户的需求，并设计出相应的方法，可能在实现中还要设计一堆的 Protected 方法，这样使得用户可以通过继承来调整默认实现。为什么这种方法不好呢？

因为考虑的过于详细，功能边界也就越大，所面对的需求也就越多，因此要提供的功能和可供用户调整的功能也就更加庞大，也就是说这种方法会使得 API 包含很多的功能，最终就是 API 膨胀性的增长。

事实上 API 中包含的功能越多，也就更加难以学习和使用，而且其学习难度往往是以几何级数进行增长，而不是线性增长。想像一下，理解并学习使用 10 个类，100 个方法的 API 对于一个程序员并不困难，大概一天就可以完成，但是对于一个 100 个类，1000 个方法的 API，即使对于一个非常优秀的程序员，估计 10 天的时间是不足以完全理解。

另外在一个庞大的 API 中，如何才能尽快的找到最重要的内容，如何找到完成所需功能的方案，一直都是 API 中设计中的一个难题。

JavaDoc 工具给用户带来了许多的方便，但一直以来它都没有解决如何学习和使用一个庞大 API 库的方法。JavaDoc 将指定包中的所有类都放置在一起，并且将一个类中的所有方

法放置在一起（译注：这里指的是 allclasses-frame.html，index-all.html），纵然是天才，看到成千上万的方法和类也只能抱头而泣了。现在看来，只能寄希望于JSR260 标准，希望它能够有效地增强 JavaDoc 工具，从而可以获得 API 的完整视图，能够更加宏观地表示 API，如果这样，即使是很庞大的 API 包，也不会显得拥挤，从而也就更加容易理解和使用。

另外 API 越大，出现的错误可能性也就越多，与前面使用的难度一样，错误的数量也是呈几何级数增长而不是线性增长。对于小的 API，投入相同的人力进行编码和测试时，可以获得更好的产出。

如果设计的 API 过于庞大，必然会包含了许多不必要的方法，至少有许多 public 的类和方法，对于大部分用户是用不到，也会占用更多的内存，并降低运行效率。这违反了通常的一个设计原则："不要让用户为他使用不到的功能付出代价"。

正确的解决方案是在现在的例子上来设计 API（译注：更象原型演化的方式）。先来想像一下：一个用户要使用 API 来解决何种问题呢，并为解决这些问题添加足够的类和方法。然后将与之无关的内容全部移除，这样可以自己来检查这些 API 的有用性，这种方法还会带来一个有用的附加功能，它可以更加有效测试代码。同时也可以将这些例子与同伴分享。（译注，这个概念与测试先行是非常相似的）。

接口的功能被夸大了：

在 Java 的世界，有一些 API 的设计原则是很通用的，如尽量使用接口的方式来表达所有的 API（不要使用类来描述 API）。接口自有它的价值，但是将所有的 API 都通过接口来表示并不见得总是一个好的设计方案。在使用一个接口来描述 API 时，必须有一个足够的理由。下面给出了一些理由：

1、接口可以被任何人所实现。假设 String 是一个接口而非类，永远都无法确认用户提供的 String 实现能够遵循你希望的规则：字符串类是一个不变量；它的 hashCode 是按照一定的算法规则来返回数字；而 length 永远都不会是一个负数等。如果真的由用户来提供一个 String 的实现，可以想象代码中要加入多少异常处理代码和相关的判断语句才能保证程序的健壮性。

实践告诉我们，如果 API 完全是由接口来定义，用户在使用这些 API 时会发现不得不进行大量的强制转型（译注：个人认为，强制转型并不是因为 API 是通过接口来定义引起的，而是不好的 API 定义引起的，而且强制转型从程序的设计角度几乎是无法避免的，除非所有的子类都不添加任何新的功能，而这一点与前面的抽象类演化又是矛盾的）。

2、接口不可能拥有构造函数或者是 static 方法。如果需要接口的实例，不可能直接实例化接口，只能通过某种方式，可能是 new 也可能是通过参数传递的方式来获得一个接口的具体实现对象。当然，这个对象可能是由你来实现的，也可能是由第三方供应商开发的。如果 Integer 是一个接口而非一个类，则无法通过 new Integer(int)来构造一个 Integer 对象，可能会通过一个 IntegerFactory.newInteger(int)来获得一个 Integer 对象实例，天啊，API 变得更加复杂和难以理解了。

（译注：个人认为原文作者有些过于极端化了，因为 Java 不是一个纯粹的 API，它同时是一种语言，一个平台，所以提供的 String 和 Integer，都是作为基础类型来提供。虽然同

意作者的观点，但是作者使用的上述例子，个人认为不是很有说服力。）

3、接口无法进行演化。假设在 2.0 版本的 API 中为一个接口添加一个方法，多米诺骨牌倒了，大量直接实现了这个接口的类，根本就无法通过编译，直到实现了这个方法为止。当然可以在调用这个新方法的时候，通过捕捉 AbstractMethodError 这个异常来保证二进制的兼容性，但是如此笨重的方法，实在不是智者所为。除非告诉用户说千万不要直接实现这个接口，请先继承所提供的那个抽象类，这样做就不会有问题了，不过用户会尖锐的责问：那为什么要提供这样一个接口，不如直接提供一个抽象类算了。

4、接口是不可以被序列化的，虽然 Java 的序列化存在许多问题，但是仍然不可避免的要用到它。象 JMX 的 API 就严重依赖于序列化接口。序列化是针对序列化接口的子类来处理的，当一个可序列化的对象被反序列化时，就有会一个相同的新对象被重新创建出来。如果这个子类没有提供一个 public 构造函数，那么可能很难在程序中使用这个功能，因为只能反序列化而不能进行序列化。而且在反序列化时，只能使用接口进行强制转型。如果要序列化的内容是一个类，那就不需要提供这样的序列化接口。

当然，对于下列情况，接口还是非常有用的：

1、回调：如果功能完全由用户来实现，在这种情况下，接口显然比抽象类更加合适。例如 Runnable 接口。特别是那些通常只含有一个方法的，往往是接口，而非类（译注：最常用的就是各种 Listener）。如果一个接口中包含有大量的抽象方法，用户在实现这个接口的时候，就不得不必须实现一些空方法。所以对于有多个方法的接口，建议提供一个抽象类，这样在接口中添加新的方法，而不需要强迫用户实现新的方法。（译注：看来作者很推荐使用接口+基类的方式来编写 API，不过 Java SE 本身就是这样做的，例如 MouseAdapter,KeyAdapter 等。个人认为，如果是规范，当然最好是接口，象 J2EE 规范；如果是框架，或者是功能包，还是建议使用接口+基类的方式。所谓的回调其实是 SPI(Service Provider Interface)的一种）。

2、多重继承：在一个继承体系比较深的结构里，可以通过接口来实现多重继承。Comparable 是一个最好的例子，比如 Integer 实现了 Comparable 接口，因为 Integer 的父类是 Number，所以通过接口的方式实现了多重继承。但是在 Java 的核心类库中，这样的经典例子并不多。通常一个类实现了多个接口并不一定是一个好的设计，因为这往往将许多责任强加在一个类上，有违基本的设计原则，而且很容易产生重复代码。如果真的需要这样一个功能，使用一个匿名类或者是一个内部类来实现这些接口，或者使用一个抽象类作为基类也是不错的方案。

3、动态代理：价值不可估量的动态代理类 java.lang.reflect.Proxy class 可以在运行的时候根据接口生成实现的内容。它将对一个接口的调用转换成对某一个对象具体方法的调用，非常的灵活，可以有效的减少代码重复。但是对于一个抽象类，就不可能动态生成一个代理对象了。如果喜欢使用动态代理技术，那么使用接口对软件开发是非常有效的。(CGLIB 有时可以有效地对抽象类实现动态代理，但是有许多限制，而且其文档也较少。）

谨慎的分包：

Java 在控制类和方法的可见性上，所支持的方式实在乏善可陈，除了

public,proteced,private 以外，就只能通过 pakcage 来控制。如果一个类或者方法想让外部的包可见，则所有的类和方法都可以访问它了，不能指定外部哪些类可以访问自身。这就意味着如果将 API 分成若干个包进行发布，则必须对这些包详细设计，避免减少 API 的公开性。

最简单的方法当然是把所有的 API 放在一个包中，这样很容易通过 package 来降低访问性。如果 API 不超过 30 个类，这个方案简直是完美。

但事事往往不尽如人事，经常 API 非常大，不适合放在一个包中。这时候可能要不得不进行私有分包了（这里的私有与 private 不一样的，只是一种伪私有），私有只是不在 JavaDoc 中输出这些类的文档信息。如果查看 JDK，会发现许多以 sun.*或者 com.sun.*打头的包相关的文档信息并没有包含在 JDK 的 JavaDoc 中。如果开发人员主要通过 JavaDoc 来使用 API，那可能根本不会注意到这些包的存在，只有查看源代码或者分析 API 的人才能看到这些 API 内容。即使发现了这些没有通过文档公开的类，也不建议使用它们，因为不通过文档公开的 API，往往也意味着它可能会随着时间的改变进行演化，也有可能在演化的过程中不能保持兼容性。（译注：象 C#支持 assembly 的访问机制，个人就感觉很好，象 Osgi 支持 Bundle，允许定制输出类也是不错误的解决方案，不过前者是语言级，而后者是框架级。）

还有将包隐藏起来的一个方式就是在包的名称中包含 internal。所以 Banana 的 API 可能会有公开的包 com.example.banana, com.example.banana.peel，也可能还有 s com.example.banana.internal 和 com.example.banana.internal.peel。

别忘记所谓的私包同样是可以访问的，更多时候这样的私包只是出于安全的考虑，建议用户不要随便访问，并没有任何语言级的约束。还有一些技术可以解决这个问题，例如 NetBeans 的 API 教程中就给出了一种解决方案。在 JMX 的 API 中，则使用了另外一种方式。象 javax.management.JMX 这个类，就只提供了 static 方法而没有提供 public 构造函数。这也就意味着你不能实例化这样一个类。（译注：不明白这个例子的意义。）

下面在设计 JMX 时的一些技巧

不变类是一个很好的设计，如果一个类可以设计成不变类，就不要用可变类！如果详细了解这样设计的原因，请参见《Effective Java》的第十三条。如果没有读过这本书，很难设计出好的 API。

另外字段信息应该是 private 的，只有 static 和 final 修饰的字段信息才能变成 public，允许外部访问。这一条是一个非常基础的原则，这里提到这个原则，只是因为在早期的 API 设计时，有些 API 违反了这个原则，这里不再给出一个例子了。

避免奇怪的设计。对于 Java 代码，已经有了许多约定俗成的方法了，如 get/set 方法，标准的异常类。即使觉得有了更好的方法，也尽量避免使用这些方法。如果使用了一些奇怪的方法名称，这样使用 API 的用户必须学习新的内容，不能按照原有的习惯来理解代码，会增加学习成本，也会增加误用的可能。

再举个例子，象 java.nio 以及 java.lang.ProcessBuilder 就是一个不好的设计，它不使用 getThing()和 setThing()方法这种方式，而使用了 thing()和 thing(T)这两个方法。许多人认为这是一个不错的设计方法，但是这样违反了常用的方法设计原则，强迫用户来学习这种API。（译注：java.nio 和 java.lang.ProcessBuilder 是指 JDK6 中的包，害得我在 JDK1.4 中找了半天，参见 http://download.java.net/jdk6/doc/api/java/lang/ProcessBuilder.html，这里所谓的 thing

和 Thing 也不是真有这个方法和类，而是 ProcessBuilder 中的 command 和 command(List)等多个方法。）

不要实现 Cloneable, 即使想某一个类支持对象的复制，这个接口也没有太多的价值，如果真想支持复制功能，提供一个复制构造函数或者是一个 static 方法来复制对象，又或者提供一个 static 的工厂方法来创建对象，也会更加有效。例如想让 Banana 这个类拥有 clone 的能力，可以使用代码如下：

```
public Banana(Banana b) {          // copy constructor
        this(b.colour, b.length);
    }
    // ...or...
    public static Banana newInstance(Banana b) {
        return new Banana(b.colour, b.length);
    }
```

构造函数的优点就在于子类可以调用父类的构造函数。static 函数则是可以返回具体类的子类实现。

《Effective Java》书中第十条则给出了 clone()带来的痛苦。

（译注：个人不同意这个观点，我觉得 clone 非常有用，特别是在多线程的处理中，我会再撰写关于 clone 方面的文章，而且前面提到的缺点也都是可以通过一些设计上的技巧来改正。）

异常应该尽可能的是 unchecked 类型的，《Effective Java》书中第 41 条则给出了详细的说明。如果当前 API 只能抛出异常，而且开发人员可以对异常进行处理，如释放资源，就可以使用 Checked 异常。因此所谓的 Checked 异常就是 API 内部与外部开发人员进行问题交互的一种方式。如网络异常，文件异常或者是 UI 异常等信息。如果是输入参数不合法，或者是一个对象的状态不正确，就应该使用 Unchecked 异常。

一个类如果不是抽象类，就应当是 final 类不可被继承。《Effective Java》第 15 章给出了足够的理由，同时也建议每个方法在默认情况下都应该是 final（目前 Java 正好相反）（译注：这点我也赞成，觉得方法默认为 final 更好，但是目前 Java 发展到当前情况下，已经不可能大规模的更改了，不能不说是 Java 语言的一个遗憾之处，C#这一点处理的更好，默认为 final，后面的留言也提到这个了）。如果一个方法可以被覆盖，一定要在文档中清楚的描述这个方法被覆盖后带来的后果，最好还能提供一些例子程序进行演示以避免开发人员误用。

总结：

- 设计需要演化，否则会降低它的价值。
- 先保证 API 的正确性，在此基础上再追求简单和高效
- 接口并不如想像中的那么有用。
- 谨慎分包可以带来更多的价值。
- 不要忘记阅读《Effective Java》（译注：难道作者和 Josh Bloch's 有分赃协议不成。）

以下是当前文章一些讨论的意见（因为比较多，所以我没有全部翻译，但是国外技术论坛上的一些讨论，其价值往往比文章的价值更大，建议可以自行阅读一下。）：

Gregor Zeitlinger 写道：

如果使用作者给出的 API 标准，C#很多方面是不是做的更好呢。

1. C#的方法在默认情况是 final 的，不可被重载。（译注：个人意见，在这一点上 C# 比 Java 更好，而且，参数默认就应该是 final，因为 java 参数是传值的，所以也不应该改变）。
2. C#只有 unchecked exception（新的规范中又重新提出了 checked exception）。（译注：个人认为 checked exception 的价值还是很大的，不过在 Java 中，有些被误用了）。

Eamonn McManus 写道：

我个人不认为让大部分方法都成为 final 是一个好的设计方案。根据我个人的经验，这种处理方式将会严重的降低代码的复用度。

如果更极端的说一下：也许不应该有 private 方法，所以的方法都应该是 protected 甚至是 public，这样可以有益于复用度。当然这样处理带来的一个明显缺点就是没有人知道哪个方法可以被安全的复写（overrid）。

（译注：这也太极端了，如果这样，一个 API 的规模恐怕会是原来的 10 倍以上，不要说复用，恐怕怎么用我都不知道了，想想一下一个 1000 个类，10000 个 public 方法的 API 包吧）。

Gregor Zeitlinger 写道：

在什么情况下我们要同时提供接口和抽象类呢。举个例子，Java 的集合框架中提供了 List 这个接口和 AbstractList 这个抽象类。

接口的好处在于

1. 多继承
2. 其实现子类可以有最大的灵活性
3. 能够将 API 的描述信息与其实现彻底分离

类的好处在于：

1. 提供通用的方法，避免重复代码
2. 能够支持接口的演化

Eamonn McManus 写道：

Bloch 在《Effective Java》中强烈建议在提供一个接口的同时，尽量提供一个实现了该接口的抽象基类。这话在 Java 集合框架的设计体现的淋漓尽至，他老兄就是 Collection 框架的主设计师。这个设计的模式在许多场景中都是非常有用的，不过也不要把它当作金科玉律，一言而敝之：不要为模式而模式。

即使你使用了接口+基类的方式，也不能保证你 API 的演化，除非你只在基类中添加方法，而不在接口中添加方法，这种情况带来的坏处就是混乱，如果一个类想调用这个新添加的方法，因为接口中没有添加这个方法，所以通过接口是无法调用的，那么只能将它强行转型，然后再调用，但有时候又很难确认你的强行转型是正确，糟糕的 ClassCastException 又出现了。除非你能保证所有的子类都继承这个基类，不过这种情况和中彩票的机会相差不多吧。

现在来谈一下 unchecked exceptions 和 C#的问题，许多人都觉得在 Java 中 Checked exceptions 并不是一个缺陷，或者说它不是一个严重的问题。但我不这样认为：象 IOException 这种异常，应该是 Checked Exception，以便由编译器来提醒程序员时要正确处理资源问题，这是一件好事，但是在 Java 中，有大量不必要的异常成为 Checked Exception，这些 Checked Exception 却给程序员带来了许多麻烦。

Robert Cooper 写道：

> 即使你使用了接口+基类的方式，也不能保证你 API 的演化。

我认为，如果可能的话，在为基类添加新方法的同时，也应该在接口中添加新的方法。我向来如此，也没有出现过什么问题。

很清楚的一点就是，如果这样做了，接口的定义改了，如果一个类是直接实现这个接口，就需要实现所有的方法。

Michael Feathers 写道：

>在什么环境下我们要同时提供接口和抽象类呢。

>接口+基类并不是可以用于所有的场景!

大部分情况下，我都会使用接口+基类这种方式，不过这种方式也会带有几个缺点。

如果你的 API 比较复杂，很难找到一个准确的入口点来使用你的 API。比如说我需要一个 IX，但是我要一步步的向下查找 AbstractX，以及相关的实现，这种接口+基类的方式加深了继承的层次，增加了 API 的复杂度。