09讲理论六：为什么基于接口而非实现编程有必要为每个类都定义接口吗

在上一节课中，我们讲了接口和抽象类，以及各种编程语言是如何支持、实现这两个语法概念的。今天，我们继续讲一个跟“接口”相关的知识点：基于接口而非实现编程。这个原则非常重要，是一种非常有效的提高代码质量的手段，在平时的开发中特别经常被用到。

为了让你理解透彻，并真正掌握这条原则如何应用，今天，我会结合一个有关图片存储的实战案例来讲解。除此之外，这条原则还很容易被过度应用，比如为每一个实现类都定义对应的接口。针对这类问题，在今天的讲解中，我也会告诉你如何来做权衡，怎样恰到好处地应用这条原则。

话不多说，让我们正式开始今天的学习吧！

如何解读原则中的“接口”二字？

“基于接口而非实现编程”这条原则的英文描述是：“Program to an interface, not an implementation”。我们理解这条原则的时候，千万不要一开始就与具体的编程语言挂钩，局限在编程语言的“接口”语法中（比如Java中的interface接口语法）。这条原则最早出现于1994年GoF的《设计模式》这本书，它先于很多编程语言而诞生（比如Java语言），是一条比较抽象、泛化的设计思想。

实际上，理解这条原则的关键，就是理解其中的“接口”两个字。还记得我们上一节课讲的“接口”的定义吗？从本质上来看，“接口”就是一组“协议”或者“约定”，是功能提供者提供给使用者的一个“功能列表”。“接口”在不同的应用场景下会有不同的解读，比如服务端与客户端之间的“接口”，类库提供的“接口”，甚至是一组通信的协议都可以叫作“接口”。刚刚对“接口”的理解，都比较偏上层、偏抽象，与实际的写代码离得有点远。如果落实到具体的编码，“基于接口而非实现编程”这条原则中的“接口”，可以理解为编程语言中的接口或者抽象类。

前面我们提到，这条原则能非常有效地提高代码质量，之所以这么说，那是因为，应用这条原则，可以将接口和实现相分离，封装不稳定的实现，暴露稳定的接口。上游系统面向接口而非实现编程，不依赖不稳定的实现细节，这样当实现发生变化的时候，上游系统的代码基本上不需要做改动，以此来降低耦合性，提高扩展性。

实际上，“基于接口而非实现编程”这条原则的另一个表述方式，是“基于抽象而非实现编程”。后者的表述方式其实更能体现这条原则的设计初衷。在软件开发中，最大的挑战之一就是需求的不断变化，这也是考验代码设计好坏的一个标准。**越抽象、越顶层、越脱离具体某一实现的设计，越能提高代码的灵活性，越能应对未来的需求变化。好的代码设计，不仅能应对当下的需求，而且在将来需求发生变化的时候，仍然能够在不破坏原有代码设计的情况下灵活应对。**而抽象就是提高代码扩展性、灵活性、可维护性最有效的手段之一。

如何将这条原则应用到实战中?

对于这条原则，我们结合一个具体的实战案例来进一步讲解一下。

假设我们的系统中有很多涉及图片处理和存储的业务逻辑。图片经过处理之后被上传到阿里云上。为了代码复用，我们封装了图片存储相关的代码逻辑，提供了一个统一的AliyunImageStore类，供整个系统来使用。具体的代码实现如下所示：

```
public class AliyunImageStore {
  //...省略属性、构造函数等...

  public void createBucketIfNotExisting(String bucketName) {
    // ...创建bucket代码逻辑...
    // ...失败会抛出异常..
  }

  public String generateAccessToken() {
    // ...根据accesskey/secrectkey等生成access token
  }

  public String uploadToAliyun(Image image, String bucketName, String accessToken) {
    //...上传图片到阿里云...
    //...返回图片存储在阿里云上的地址(url) ...
  }

  public Image downloadFromAliyun(String url, String accessToken) {
    //...从阿里云下载图片...
  }
}

// AliyunImageStore类的使用举例
public class ImageProcessingJob {
  private static final String BUCKET_NAME = "ai_images_bucket";
  //...省略其他无关代码...

  public void process() {
    Image image = ...; //处理图片，并封装为Image对象
    AliyunImageStore imageStore = new AliyunImageStore(/*省略参数*/);
    imageStore.createBucketIfNotExisting(BUCKET_NAME);
    String accessToken = imageStore.generateAccessToken();
    imagestore.uploadToAliyun(image, BUCKET_NAME, accessToken);
  }

}
```

整个上传流程包含三个步骤：创建bucket（你可以简单理解为存储目录）、生成access token访问凭证、携带access token上传图片到指定的bucket中。代码实现非常简单，类中的几个方法定义得都很干净，用起来也很清晰，乍看起来没有太大问题，完全能满足我们将图片存储在阿里云的业务需求。

不过，软件开发中唯一不变的就是变化。过了一段时间后，我们自建了私有云，不再将图片存储到阿里云了，而是将图片存储到自建私有云上。为了满足这样一个需求的变化，我们该如何修改代码呢?

我们需要重新设计实现一个存储图片到私有云的PrivateImageStore类，并用它替换掉项目中所有的AliyunImageStore类对象。这样的修改听起来并不复杂，只是简单替换而已，对整个代码的改动并不大。不过，我们经常说，“细节是魔鬼”。这句话在软件开发中特别适用。实际上，刚刚的设计实现方式，就隐藏了很多容易出问题的“魔鬼细节”，我们一块来看看都有哪些。

新的PrivateImageStore类需要设计实现哪些方法，才能在尽量最小化代码修改的情况下，替换掉AliyunImageStore类呢？这就要求我们必须将AliyunImageStore类中所定义的所有public方法，在PrivateImageStore类中都逐一定义并重新实现一遍。而这样做就会存在一些问题，我总结了下面两点。

首先，AliyunImageStore类中有些函数命名暴露了实现细节，比如，uploadToAliyun()和downloadFromAliyun()。如果开发这个功能的同事没有接口意识、抽象思维，那这种暴露实现细节的命名方式就不足为奇了，毕竟最初我们只考虑将图片存储在阿里云上。而我们把这种包含“aliyun”字眼的方法，照抄到PrivateImageStore类中，显然是不合适的。如果我们在新类中重新命名uploadToAliyun()、downloadFromAliyun()这些方法，那就意味着，我们要修改项目中所有使用到这两个方法的代码，代码修改量可能就会很大。

其次，将图片存储到阿里云的流程，跟存储到私有云的流程，可能并不是完全一致的。比如，阿里云的图片上传和下载的过程中，需要生产access token，而私有云不需要access token。一方面，AliyunImageStore中定义的generateAccessToken()方法不能照抄到PrivateImageStore中；另一方面，我们在使用AliyunImageStore上传、下载图片的时候，代码中用到了generateAccessToken()方法，如果要改为私有云的上传下载流程，这些代码都需要做调整。

那这两个问题该如何解决呢？解决这个问题的根本方法就是，在编写代码的时候，要遵从“基于接口而非实现编程”的原则，具体来讲，我们需要做到下面这3点。

1. 函数的命名不能暴露任何实现细节。比如，前面提到的uploadToAliyun()就不符合要求，应该改为去掉aliyun这样的字眼，改为更加抽象的命名方式，比如：upload()。
2. 封装具体的实现细节。比如，跟阿里云相关的特殊上传（或下载）流程不应该暴露给调用者。我们对上传（或下载）流程进行封装，对外提供一个包裹所有上传（或下载）细节的方法，给调用者使用。
3. 为实现类定义抽象的接口。具体的实现类都依赖统一的接口定义，遵从一致的上传功能协议。使用者依赖接口，而不是具体的实现类来编程。

我们按照这个思路，把代码重构一下。重构后的代码如下所示：

```
public interface ImageStore {
  String upload(Image image, String bucketName);
  Image download(String url);
}

public class AliyunImageStore implements ImageStore {
  //...省略属性、构造函数等...

  public String upload(Image image, String bucketName) {
    createBucketIfNotExisting(bucketName);
    String accessToken = generateAccessToken();
    //...上传图片到阿里云...
    //...返回图片在阿里云上的地址(url)...
  }
```

```
  public Image download(String url) {
    String accessToken = generateAccessToken();
    //...从阿里云下载图片...
  }

  private void createBucketIfNotExisting(String bucketName) {
    // ...创建bucket...
    // ...失败会抛出异常..
  }

  private String generateAccessToken() {
    // ...根据accesskey/secrectkey等生成access token
  }
}

// 上传下载流程改变：私有云不需要支持access token
public class PrivateImageStore implements ImageStore  {
  public String upload(Image image, String bucketName) {
    createBucketIfNotExisting(bucketName);
    //...上传图片到私有云...
    //...返回图片的url...
  }

  public Image download(String url) {
    //...从私有云下载图片...
  }

  private void createBucketIfNotExisting(String bucketName) {
    // ...创建bucket...
    // ...失败会抛出异常..
  }
}

// ImageStore的使用举例
public class ImageProcessingJob {
  private static final String BUCKET_NAME = "ai_images_bucket";
  //...省略其他无关代码...

  public void process() {
    Image image = ...;//处理图片，并封装为Image对象
    ImageStore imageStore = new PrivateImageStore(...);
    imagestore.upload(image, BUCKET_NAME);
  }
}
```

除此之外，很多人在定义接口的时候，希望通过实现类来反推接口的定义。先把实现类写好，然后看实现类中有哪些方法，照抄到接口定义中。如果按照这种思考方式，就有可能导致接口定义不够抽象，依赖具体的实现。这样的接口设计就没有意义了。不过，如果你觉得这种思考方式更加顺畅，那也没问题，只是将实现类的方法搬移到接口定义中的时候，要有选择性的搬移，不要将跟具体实现相关的方法搬移到接口中，比如AliyunImageStore中的generateAccessToken()方法。

总结一下，我们在做软件开发的时候，一定要有抽象意识、封装意识、接口意识。在定义接口的时候，不要暴露任何实现细节。接口的定义只表明做什么，而不是怎么做。而且，在设计接口的时候，我们要多思考一下，这样的接口设计是否足够通用，是否能够做到在替换具体的接口实现的时候，不需要任何接口定义的改动。

是否需要为每个类定义接口?

看了刚刚的讲解，你可能会有这样的疑问：为了满足这条原则，我是不是需要给每个实现类都定义对应的接口呢？在开发的时候，是不是任何代码都要只依赖接口，完全不依赖实现编程呢?

做任何事情都要讲求一个“度”，过度使用这条原则，非得给每个类都定义接口，接口满天飞，也会导致不必要的开发负担。至于什么时候，该为某个类定义接口，实现基于接口的编程，什么时候不需要定义接口，直接使用实现类编程，我们做权衡的根本依据，还是要回归到设计原则诞生的初衷上来。只要搞清楚了这条原则是为了解决什么样的问题而产生的，你就会发现，很多之前模棱两可的问题，都会变得豁然开朗。

前面我们也提到，这条原则的设计初衷是，将接口和实现相分离，封装不稳定的实现，暴露稳定的接口。上游系统面向接口而非实现编程，不依赖不稳定的实现细节，这样当实现发生变化的时候，上游系统的代码基本上不需要做改动，以此来降低代码间的耦合性，提高代码的扩展性。

从这个设计初衷上来看，如果在我们的业务场景中，某个功能只有一种实现方式，未来也不可能被其他实现方式替换，那我们就没有必要为其设计接口，也没有必要基于接口编程，直接使用实现类就可以了。

除此之外，越是不稳定的系统，我们越是要在代码的扩展性、维护性上下功夫。相反，如果某个系统特别稳定，在开发完之后，基本上不需要做维护，那我们就没有必要为其扩展性，投入不必要的开发时间。

重点回顾

今天的内容到此就讲完了。我们来一块总结回顾一下，你需要掌握的重点内容。

1.“基于接口而非实现编程”，这条原则的另一个表述方式，是“基于抽象而非实现编程”。后者的表述方式其实更能体现这条原则的设计初衷。我们在做软件开发的时候，一定要有抽象意识、封装意识、接口意识。越抽象、越顶层、越脱离具体某一实现的设计，越能提高代码的灵活性、扩展性、可维护性。

2.我们在定义接口的时候，一方面，命名要足够通用，不能包含跟具体实现相关的字眼；另一方面，与特定实现有关的方法不要定义在接口中。

3.“基于接口而非实现编程”这条原则，不仅仅可以指导非常细节的编程开发，还能指导更加上层的架构设计、系统设计等。比如，服务端与客户端之间的“接口”设计、类库的“接口”设计。

课堂讨论

在今天举的代码例子中，尽管我们通过接口来隔离了两个具体的实现。但是，在项目中很多地方，我们都是通过下面第8行的方式来使用接口的。这就会产生一个问题，那就是，如果我们要替换图片存储方式，还是需要修改很多类似第8行那样的代码。这样的设计还是不够完美，对此，你有更好的实现思路吗?

```
// ImageStore的使用举例
public class ImageProcessingJob {
  private static final String BUCKET_NAME = "ai_images_bucket";
  //...省略其他无关代码...

  public void process() {
    Image image = ...;//处理图片，并封装为Image对象
    ImageStore imageStore = new PrivateImageStore(/*省略构造函数*/);
    imagestore.upload(image, BUCKET_NAME);
  }
```

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

精选留言

zeta

其实这篇和上一篇可以讲的更好的。首先，我反对接口是has-a的说法，我坚持接口的语义是behaves like(这个其实我也是在某一本书上看的). 咱们看下哪个更通顺和达意，A AliyunImageStorage has a DataStorage. or A AliyunImageStorage behaves like a DataStorage? 除非你在第一句加上 A AliyunImageStorage has some behaviors of DataStorage. 但这基本也就是behaves like的意思了。

第二，我觉得咬文嚼字的确没有什么意义，但为什么说上述话题，难道讲接口的例子不用出现接口多重继承么，引用我之前留言：拿一个C++中举的多重继承例子来说，吸血鬼分别继承自蝙蝠和人，那么吸血鬼is a蝙蝠么？吸血鬼is a人么？所以其实两个都不是，这就是设计上的语义问题。这里缺失了除了is a的另一个概念，behaves like，也就是多重继承的真义实际上是behaves like，也就是接口的意义。A vampire behaves like humans and bats. 而这是接口能多重的原因，一个类可以具有多重行为，但是不能是多种东西。

所以其实也就是说，只有当前模块涉及到抽象行为的时候，才有必要设计接口，才有可能利用接口多重继承的特性来更好的将各种行为分组。

2019-11-22 19:43

helloworld

到目前为止老师所讲理的的理论都懂～至于思考题用简单工厂，反射等方式感觉都不行。给老师提个小小的建议：能不能和隔壁的『MySQL实现45讲』的专栏一样在下一节课程的末尾集中回答一下上一节课程的课后习题？感谢

2019-11-23 11:20

香蕉派2号

思考题

解决方案=配置文件+反射+工厂模式

2019-11-22 05:40

业余爱好者

关于抽象和函数命名的问题，不知道哪个大佬说过这么一句话:

每个优秀的程序员都知道，不应该定义一个attackBaghdad() ‘袭击巴格达‘ 的方法，而是应该把城市作为函数的参数 attack(city)。

2019-11-22 07:43

辣么大

关于思考题我想出两种方法改进：简单工厂方法和使用反射。

1、简单工厂方法

ImageStore imageStore = ImageStoreFactory.newInstance(SOTRE_TYPE_CONFIG);

config文件可以写类似properties的文件，使用key-value存储。

缺点：再新增另一种存储手段时，需要修改工厂类和添加新的类。修改工厂类，违反了开放-封闭原则。

那有没有更好一点的方法呢?

2、使用反射。
在配置文件中定义需要的image store类型。
在ProcessJob中
ImageStore store = (ImageStore) Class.forName(STORE_CLASS)
.newInstance();

缺点：使用反射，在大量创建对象时会有性能损失。

关于减少ProcessJob中的修改，还有没有更好的方法呢？我只是抛砖引玉，希望和大家一起讨论。具体实现：https://github.com/gdhucoder/Algorithms4/tree/master/geekbang/designpattern/u009

补充：
关于access token：Aliyun的AccessToken时有expireTime时限的。不需要每次重新获取，过期时重新获取即可。
2019-11-22 07:48

秋惊蛰
依赖注入，从外部构建具体类的对象，传入使用的地方
2019-11-22 02:21

编程界的小学生
首先这篇文章受益匪浅，尤其是第二点，与特定实现有关的方法不要暴露到接口中，深有体会。

其次问题解答
我个人的解决方案是这种情况不要去直接new，而是用工厂类去管理这个对象，然后名字可以起成getInstance这类不包含某个具体实现的含义的抽象名称。将来修改直接修改工厂类的getInstance方法即可，这种方式可取吗？还有其他更好的方式吗？求老师点评。
2019-11-22 00:14

守拙
课堂讨论answer:
考虑使用工厂模式生成ImageStore实例.这样就可以将调用者和具体ImageStore解耦.

例:

```
public class ImgStoreFactory {

private ImgStoreFactory(){

}

public static ImageStore create(Class<?> clz){
if (clz == AliyunStore.class){
return new AliyunStore();
}else if (clz == PrivateYunStore.class){
return new PrivateYunStore();
}else {
throw new IllegalStateException("..");
}
}

public static void main(String[] args) {
ImageStore store = ImgStoreFactory.create(AliyunStore.class);
```

store.dosth();
}
}

另外有一点不太同意作者的说法:
上节课作者将Contract翻译为"协议",
我认为是不恰当的.
在计算机领域, 通常使用Protocol代表协议.
个人认为Contract更恰当的翻译是"契约".
2019-11-22 17:52


失火的夏天
思考题估计就是要引出工厂模式了吧
2019-11-22 00:16


雷霹雳的爸爸
要不是有一开始的课程大纲，我以为课堂讨论是要启发大家，在下节就要讲创建型模式，工厂模式，工厂方法什么的了

但转念一想，这想法或许太肤浅了，毕竟大多数创建型方法都有一个明显的对具体类型的依赖（这里先预先排除抽象工厂，觉得有点小题大做这样搞），都不是一个最终能让人感到内心宁静的做法

这节既然讲的是依赖于抽象而不是依赖于具体，那比较得瑟的玩儿法恐怕应该是直接在ImageProcessingJob类和ImageStore接口这两个类型关系上充分体现出依赖倒置的思路，把最后一点执行创建ImageStore类型实例的痕迹彻底关在ImageProcessingJob的门外，虽然必然得有人去考虑实际至少调一下ImageStore具体类型实例的这个创建过程，但ImageProcessingJob这爷是不打算操心这事了，它只需要留个口子，让别人把ImageProcessingJob放到自己锅里，自己就可以开始炒菜了

也就是从形式上，ImageProcessingJob这个类只需要保留对ImageStore接口的依赖就可以了，具体留口子的手段则要考虑依赖注入，形式上有两种：

+ 一种是可能更OO样子的一点，即声明一个ImageStore的field在ImageProcessingJob类里面
- 如果说有什么好处，恩，可以理解为能对客户程序隐藏了ImageStore类型的信息，是的，连类型信息都隐藏掉；好吧，还是得关心别人，毕竟这世界上不是仅有自己一个
- 具体操作起来，由于不能声明field时候直接new，要不又变回去了，但又不能NPE吧，所以不考虑创建，我还是得考虑怎么把实例请进来，就是上面说的至少留个口子
- 这时候可能不得不借助依赖注入的帮助了（否则就是依赖查找，还是工厂），即
- 通过ImageProcessingJob的构造函数注入或者利用field注入来获取ImageStore接口的实例，或者ImageProcessingJob如果依赖项多，Builder一下也很好
- 毕竟ImageProcessingJob这个类型在我们讨论的上下文里面是如此具体的一个类，就不过分追溯它的创建责任及执行在哪里了
+ 另一种，表面粗暴直接看似问题多多，但是细品也有点意思的，那就是process方法直接增加一个ImageStore的参数就完了，OMG我在干什么
- 没有B方案的设计自身无法证明自己更好
- 相对于上面的，直接的问题是会对process方法直接依赖的客户程序会和ImageStore这个类型产生耦合
* 如果客户程序是一个类（还能是什么？），要么有一个field等着inject进来，要么是通过调用process的method传进来，要么就是无中生有（直接new了或用创建型模式）
* 这都可能造成没有充分的设计隔离，至少让客户程序造成信息冗余，承担了不必要的职责等问题
- 但事实上也不是完全没好处，这种灵活性体现在它没有把ImageStore的逻辑固化在任一个ImageProcessingJob实例里面
* 考虑上面第三类无中生有的方式，假设是创建型的工厂方法或类似手段，则可以提供对method参数（业务层面的动态输入，例如最终操作用户的提供的值）的响应能力
* 这和，执行排序算法骨架确定，但是需要给定两个元素（复杂对象）比较规则这种思路有相似之处，毕竟我需要的是对方的能力而不是对方的数据或者数据视图，这时候这么做还是很有诱惑力的
- 如果脱离开场景，实际上这种动态性还更强，但问题就在于这种动态性会不会对于具体场景有价值
- 从这个实例上看，也许没这么明显，因为不同的对象存储后端更有可能是环境（测试、生产？但12 factor让我们...好歹测试环

境还是也上云吧）不同造成的，而非基于动态的用户信息输入
- 但，事无绝对吧，假设，用户有选择我要针对具体这一张，我特么上传那一刻选择一个存储后端的需求
- 然后为了方便用户，用户竟然可以勾选，以后使用同样地选择...
- 我觉得除了脑子进水的犬类应该没人会干这种没问题制造问题也要上的方案吧

所以综上所属，还是field一个ImageStore接口来搞吧

这极客时间也让人想吐槽，能敲2000字，结果就只留这么大点儿一个输入框...你要不就限制200字我还能少敲一点...我这写的兴起还得外面写完了贴过来...

2019-11-22 16:24

William

所以思考题，想到的是，将接口作为构造函数中的参数，传递进来，再调用.

2019-11-22 01:39

Monday

依赖注入可以解决思考题，基于接口的实现有多种时，注入处也需要指明是哪吒实现

2019-11-22 08:50

YouCompleteMe

抽象工厂，把创建具体类型放到工厂类里

2019-11-22 01:50

程斌

存储图片的方式写入到配置文件，第8行改用传入类型参数来实例化不同的对象，明天补上代码。

2019-11-22 00:20

二星球

使用策略模式，在建一个Context类，使用聚合持有这个接口实例引用，其它所有地方都用这个context类，变动的时候，只变这个context类就行了，其它不动

2019-11-23 14:53

Milittle

再说一句 面向接口编程的精髓 我的理解是我们在使用接口的时候 关心我们要做什么 而不是怎么做 怎么做都封装在具体实现类中。而且最主要的是 接口抽象

2019-11-22 16:31

NoAsk

关于什么时候定义接口的一些拙见：
当方法会有其他实现，或者不稳定的时候需要定义接口；
1.不稳定的方法一般能事先确定，用接口能提高可维护性
2.但在开发时往往不确定是否需要其他实现，我的原则是等到需要使用接口的时候再去实现。所以根据kiss原则一般我会先用方法实现，如果有一天真的需要有新的实现的时候再重新抽象出接口对代码进行小重构。

就老师的例子进行一下说明：
刚开始只需要阿里云进行图片上传下载功能，我就先只实现阿里云的图片上传下载方法。
后期发现需要有私有云的上传下载方法的话，那就对这个功能通过接口进行抽象。但是你永远不知道到底是新的图片上传下载功能先来到还是其他阿里接口先来到，如果是新的阿里接口，也是用的这一套token方法，那就用抽象方法或接口对token部分实现抽象。

课后问题：
简单工厂，工厂模式可以提高可扩展性，维护性。
java spring项目可以使用注入的方式。

2019-11-22 07:14

bearlu

老师，希望能把示例代码和问题代码也放到Github上。

2019-11-22 08:52

作者回复

我抽空整理一下放上去

https://github.com/wangzheng0822

2019-11-22 09:15

超威丶

个人觉得维护map是最好的选择，实现类型和具体实现对应。

2019-11-22 08:21

Paul Shan

基于抽象而非具体体现了信息隐藏和分离代码中稳定性不同的部分。在一个上传图片的部分不需要知道图片是如何上传的，阿里云以及token就属于过多的信息，有必要隐藏这些信息。另外一方面上传图片这件事比阿里云实现要稳定的多，不上传图片的概率低于不用阿里云上传图片的概率。这里有必要分离图片上传这个接口和用阿里云上传这个实现。

不过原来的实现也没什么问题，毕竟谁也不能未卜先知，将来一定会替换阿里云。如果我拿到这个变更需求，我会先用同名接口替换原来的实现（原来的阿里类实现清晰，功能单一，只是不适合直接调用），然后用adapter来转接口，然后一步一步实现接口和实现的分离，目标是接口能够隐藏信息，实现能够清晰明了，每一步都能用IDE工具重构，每一步都能编译和测试。

2019-11-22 04:55