<https://www.zhihu.com/question/25654738/answer/31302541>

归纳题主的问题：

这个世界上有各种各样的框架，设计这些五花八门框架的初衷到底是什么？我们该不该学习框架，该如何学习使用这些框架？

回答题主的问题：

一、首先，到底什么是框架？

想要回答这个问题，我们要慢慢来。

①

首先从DRY原则开始说起

Don't Repeat Yourself，不要重复你的代码。

DRY原则的重要性怎么提都不过分，很多人说编程是种机械性的工作，而有很多程序员也自嘲为码农，意为编程成了一种没有技术含量的体力性工作。如果不想沦为这个境界，首先需要的就是将DRY原则融入你的血液，在今后的编码工作中加以运用。

1）最初级的DRY：语法级别

System.out.println(1);
System.out.println(2);
……
System.out.println(10);

我想只要学过基础语法，都会采用下面的形式。

for (int i = 1; i <= 10; i++) {
System.out.println(i);
}

如果发现有任何人采用上面一种形式的编码形式，那么不用怀疑，他对于编程绝对还没有入门。

我们当然会选择省力的做法，这种做法不但省力，还会有利于我们后续修改或扩展这组代码，如：

for (int i = 1; i <= 10; i++) {
System.out.println(i \* 2 + 1);
}

我们进行这样的修改，只需要修改一处，而上面的形式却需要修改10处，当然会更麻烦且更容易出错，所以请记住能不重复就不重复。

2）进阶的DRY原则：方法级别

当我们经常写一些重复性代码时，我们就要注意看能否将其抽取出来成为一个方法，如：

try {
Thread.sleep(1000);
} catch (InterruptedException e) {
e.printStackTrace();
}

让我们将其抽取到一个方法 threadSleep() 中，这样我们只需要调用 threadSleep() 就可以实现原来的功能，不但所需敲击的代码更少，而且代码看起来更加清楚明白。而为了增加这个方法的复用性，我们还可以将其中固定的数字抽取成为参数，如：

private static void threadSleep(int millis) {
try {
Thread.sleep(millis);
} catch (InterruptedException e) {
e.printStackTrace();
}
}

这样我们就可以利用这个方法实现不同时间的sleep了。要注意提高代码的复用性也是实践DRY原则的一个重要方法，在后面我们也可以看到框架为了提高所谓的灵活性进行的一些设计，如在适当的位置增加扩展点。

3）继续进阶的DRY原则：类型级别

现在我们看一个类

public class Person {
private String name;
private int age;
// Setter & Getter ...
}

我们新建一些Person类实例，并进行一些操作：

Person person = new Person();
person.setName("jack");
person.setAge(18);
Person person2 = new Person();
person2.setName("rose");
person2.setAge(17);
.....
System.out.printf("Name: %s, Age:%d\n", person.getName(), person.getAge());
System.out.printf("Name: %s, Age:%d\n", person2.getName(), person2.getAge());
.....

观察这些代码，其实有很大的DRY改造空间，首先可以添加一个构造方法

public Person(String name, int age) {
this.name = name;
this.age = age;
}

其次，可以添加一个toString()方法

public String toString() {
return String.format("Name: %s, Age: %d", name, age);
}

这样的话，上面的代码就可以改成下面的形式。

Person person = new Person("jack", 18);
Person person2 = new Person("rose", 17);
......
System.out.println(person.toString());
System.out.println(person2.toString());
......

4）继续继续进阶的DRY原则：多个类组合级别

上面的代码我们其实还是有改善空间，就是利用容器类

List<Person> list = new ArrayList<>();
list.add(new Person("jack", 18));
list.add(new Person("rose", 17));
......
list.forEach(p -> System.out.println(p));

这里利用JDK8的Stream API以及Lambda表达式输出，其实可以进一步简化为

list.forEach(System.out::println);

这里我们可以看到，基本上我们写代码只写有变化的代码，而尽量不写机械性重复性的代码，其实后面我们就会知道，这就叫专注于业务逻辑，所谓业务逻辑就是你这个项目中，与别的项目都不一样的地方，必须由你亲自去编写实现的部分。

其实容器类很大程度上也是为了帮助我们编写代码而被设计出来的，首先让我们不必为每一个对象起名字（省去了person,person2,...等变量），然后又为批量操作提供了可能性。像是这样一系列有用的类组合起来可以称之为类库。常用的类库有Commons-Lang包等，为我们提供了一大批实用方法，我之所以提到类库，也是因为框架其实也是一种特殊的类库，但是却与一般的类库有着本质的不同。

②

设计模式，更高层级的DRY应用

上面我讲到了DRY原则的几个层次，一般情况下大家也早就这样使用了，属于入门之后很容易自己就想到得一些层次。但是设计模式不一样，设计模式是经过长时间编码之后，经过系统性的总结所提出的针对某一类问题的最佳解决方案，又称之为最佳实践。

而在小规模的编码工作中，其实并不需要什么设计模式，只有大型程序才有设计模式发挥的空间，所以我们需要借助一些特定领域有足够规模的问题来了解一下设计模式存在的必要性。

1）连接数据库，进行一些操作，并安全释放数据库连接。

public static boolean updatePassword(String username, String password, String newpassword) {
Connection conn = null;
PreparedStatement stmt = null;
ResultSet rs = null;
boolean success = false;
try {
conn = beginTransaction();
stmt = conn.prepareStatement("select id, password from user where username = ?");
stmt.setString(1, username);
rs = stmt.executeQuery();
if (rs.next()) {
if (rs.getString("password").equals(password)) {
PreparedStatement stmt2 = null;
try {
stmt2 = conn.prepareStatement("update user set password = ? where id = ?");
stmt2.setString(1, newpassword);
stmt2.setLong(2, rs.getLong("id"));
success = stmt2.executeUpdate() > 0;
} finally {
safeClose(stmt2);
}
}
}
commitTransaction(conn);
return success;
} catch (SQLException e) {
rollbackTransaction(conn);
throw new RuntimeException(e);
} finally {
safeClose(rs);
safeClose(stmt);
safeClose(conn);
}
}

上面是一个简单的数据库事务，虽然只有一个查询和一个更新，但是想要将其继续简化却并不容易，虽然其中有关于业务逻辑的部分只是少量几行代码，但是初始化，异常，提交，回滚操作让我们很难抽取出一个合适的方法来。虽然我们已经抽取出了 begin,commit,rollback,safeClose等方法，但是仍嫌繁琐。

我们发现之所以我们难以抽取方法，主要是因为流程，因为里面牵扯到流程控制，而流程控制一般是由我们程序员来控制的，所以也就必然需要我们手动编码来完成。难道真的就不能继续简化了吗？这就是需要设计模式的时候了。

2）应用设计模式「模板方法模式」

public static boolean updatePassword(String username, String password, String newpassword) {
return connection(conn -> statement(conn, "select id, password from user where username = ?", stmt -> {
stmt.setString(1, username);
return resultSet(stmt, rs -> {
if (rs.next()) {
if (rs.getString("password").equals(password)) {
long id = rs.getLong("id");
return statement(conn, "update user set password = ? where id = ?", stmt2 -> {
stmt2.setString(1, newpassword);
stmt2.setLong(2, id);
return stmt2.executeUpdate() == 1;
});
}
}
return false;
});
}));
}

可以看到，所有的conn,stmt,rs的开启和关闭，事务的提交和回滚都不用自己手动编写代码进行操作了，之所以可以达到这个效果，就是因为使用了模板方法设计模式，核心就是通过回调方法传递想对资源进行的操作，然后将控制权交给另一个方法，让这个方法掌握流程控制，然后适当的时候回调我们的代码（也就是我们自己写的业务逻辑相关的代码）。

这是需要额外写的几个方法

你们可能会疑惑，这些代码加上我们写的业务逻辑的代码，比原来的代码还要长，有什么必要使用这个设计模式。这正是我前面已经指出的一个问题，那就是要你的程序规模足够大才有必要应用设计模式，试想如果你有上百个乃至上千个数据库操作方法需要写，那么是不是写这几个额外的方法，就不算什么了呢。

其实这正是DRY原则在更高层次上的应用，即结合设计模式来达到更高层次的代码复用效果，进而应用DRY原则。而想要在这个层次继续向上攀升，那就必须是结合众多设计模式以及一些高层架构设计，能够帮助我们实现这一目的的就是框架。

3）框架，是设计模式的集大成者，是DRY原则的最高应用

先让我们来看一下，使用框架会是什么样的一种体验？

这里以Hibernate + Spring声明式事务为例

@Transactional
public boolean updatePassword(String username, String password, String newpassword) {
User user = (User) session().createQuery("from User where username = :username")
.setString("username", username)
.uniqueResult();
if (user != null && user.getPassword().equals(password)) {
user.setPassword(newpassword);
return true;
}
return false;
}

可以发现令人惊讶的简洁，而且代码逻辑异常清晰，完全不需要考虑conn,stmt,rs等资源的释放，以及事务的提交和回滚，但是这些事情其实框架已经默默的帮我们做到了。这才叫真正的专注于业务逻辑，尽最大可能的只写与业务逻辑有关的代码。

当然这些框架的效果虽然神奇，其实只要细细探究其内部原理，是完全可以理解并掌握的。

二、那么问题就来了，框架到底是什么？要不要学，怎么学？

上面我说过了，框架其实就是一个或一组特殊的类库，特殊在什么地方？特殊在控制权转移！

框架与一般类库不同的地方是，我们调用类库，而框架调用我们。也就是说框架掌握整个程序的控制权，我们必须一定程度上把程序流程的控制权交给框架，这样框架才能更好的帮助我们。

下面以JavaWeb开发为例再进行一些说明，并顺便简单介绍一下JavaWeb的一些脉络。

①

静态网页时代

本来网站都是一个个静态HTML组成的，或许这些网页还是用Dreamweaver写的，但是这样的静态页面显然不能满足我们，很快我们就迎来了动态网页的时代。

②

Servlet时代

如果熟悉HTTP协议的话，我们就知道其实访问网页的过程不过是一次TCP连接罢了。浏览器发起TCP连接到服务器，服务器接受请求，然后返回HTML代码作为响应。那么我们完全可以等到接受到请求之后，再动态生成HTML代码返回给客户端。

Servlet就是这么做的，其主要代码不过是利用out.write()一点一点的输出HTML代码罢了。当然我们可以在其中掺杂一点动态的东西，如返回当前的时间。

③ JSP包打天下的时代

纯粹的Servlet很是丑陋，给前端程序员理解和修改这样的代码带来了很多困难。因此JSP技术被发明了出来，原理也不复杂，就是不直接写Servlet，而是先写好JSP文件，再由服务器将JSP文件编译成Servlet。而JSP中是以常见的HTML标签为主，这样前端程序员就能方便的修改这些代码了。

PS：由只使用 Servlet到使用JSP，虽然是一个简单的变化，但这迎合了前后端专业分工的大趋势，让前段人员只需要懂得HTML/CSS/JavaScrip代码就可以开始工作，而不需要学习Servlet那枯燥无味的用法，因此借着JSP技术的东风，JavaWeb技术迅速的扩展开来了。

④ Servlet + JSP 时代

随着JSP技术的发展，用它写成的网站也越来越大，业务逻辑也越来越复杂。开发人员渐渐发现整个网站渐渐的再次变成了一团乱麻，不仅仅是JSP中夹杂了大量的Java代码，页面之间的耦合关系也越来越紧密。

即便是要修改一个简单的按钮文本，或者是引入一段静态的内容，也需要打开越来越庞大的JSP页面，艰难到找到需要修改的部分，有时还不仅仅是一处，这种修改是有很大的风险的，完全有可能引入新的错误。

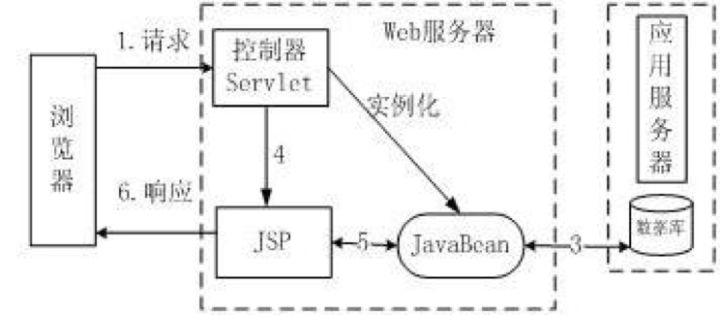
这时候开发者渐渐意识到，仅仅使用JSP是不行的，JSP承担了太多的责任。这时人们又想起了Servlet，Servlet中主要使用Java代码，处理业务逻辑非常轻松。如果JSP只使用HTML代码，而将业务逻辑的代码转移到Servlet中，就可以大大的减轻JSP的负担，并且让前后端分工更加明确。

⑤

MVC模式时代

在

Servlet + JSP模式的基础上，Java阵营进一步发展出了一种适合JavaWeb应用的设计模式，MVC设计模式，即将程序分为显示层(Viewer），控制层(Controller)，模型层(Model)。如下图所示：



一次典型的访问是这样的流程：

1. 用户输入网址或点击链接或提交表单，浏览器发起请求

2. --> 通过互联网，通过HTTP协议 -->

3. Tomcat接受到HTTP请求，生成HttpServletRequest对象，根据Web.xml的配置，调用开发者编写的HttpServlet，HttpServlet根据请求内容，调用JavaBean获取数据，JavaBean从数据库获取数据，返回HttpServlet，HttpServlet将数据转发给JSP，JSP负责将数据渲染为HTML，由Tomcat负责将HTML转化为HTTP响应，返回客户端。

4. --> 通过互联网，通过HTTP协议 -->

5. 客户端浏览器接收到HTTP响应，浏览器将HTML渲染为页面，并运行其中可能存在的JavaScript进一步调整界面。

整个流程必须由开发者精确设计才能运作流畅，其中客户端HTML和JavaScript属于前端设计，服务器运行的其他内容属于后端设计。虽然符合J2EE规范的Tomcat等应用服务器已经帮我们实现了最复杂的一块，即HTTP协议部分，还给我们提供了JSP这个模板引擎，以及自定义标签等手段。但是在控制层，在模型层，J2EE能给我们的帮助少之甚少。

就拿用户提交一个表单为例，而我们在Servlet中获取参数为例，虽然不用我们解析HTTP报文，应该已经是要谢天谢地了，但是我们要做的事情仍然很多，分析一下：

1. 客户端传过来的数据全是文本，而我们需要的是Java对象。

2. 凡是文本就有编码问题，而这需要前后端配合解决。

3. 客户端的输入是不可信的，我们必须校验参数的合法性。

4. 我们还必须将校验结果反馈给客户，并且最好不要让客户全部重新输入。

5. 我们往往不是只有一个参数需要，而是有几个甚至更多参数，要妥善的处理各种情况组合。

这些事情几乎全部都需要我们手动编码来完成，几乎每一个 Servlet 都充斥着这样的代码，设置编码，获取参数，校验参数，校验通不过返回错误信息，校验通过则进行业务处理。而更重要的是，获取参数仅仅是整个流程中的一小步，我们的Servlet中存在着大量的重复性，机械性代码，而处理业务逻辑的代码可能只有一两行。

⑥

JavaWeb框架

既然存在着大量的重复，我们当然不能忍，必须请出DRY大法。显然JavaWeb应用是一个规模庞大，流程复杂的应用，我们正需要JavaWeb框架的帮助。以Struts2框架为例，他能给我们什么帮助呢？

1. 在控制层，由Struts2的核心控制器接管控制权，将本来在Web.xml进行配置的一些工作，转移到自定义的struts.xml文件中，这个文件的配置形式更友好。

2. Struts2封装了Serlvet Api，使用POJO对象作为控制器（Action），大量使用反射，不要求继承特定类，有利于复用及单元测试。提供ActionSupport类，结合struts2标签，能很方面实现的校验信息的收集及反馈。

3. 提供国际化支持，在显示层有国际化相关的标签，在控制层由国际化相关的API。提供基于配置的校验及JS生成技术。智能化的参数类型转换，支持自定义转换器。提供Action拦截器，方便实现AOP模式。

4. 提供了基于OGNL表达式的数据共享模式，前后端数据交流更简单，提供了Struts2标签库，简单好用，支持多种模板，如FreeMarker，支持各种插件，如JSON，支持整合多种框架，如Spring。总之一句话，能在各方各面给我们强大的帮助。

⑦

所以当然要学框架，要用框架，那么要怎么学？

1. 用框架要知其然，还要知其所以然，要大体明白框架实现一个功能特性的原理，不能只是会用，只是觉得很神奇就可以了。就拿前面的Hibernate + Spring声明式事务为例，要弄明白框架这部分是怎么实现的。

2. 首先要夯实你的语言基础，如JavaSE基础，语法掌握，用法掌握，有些同学语法还不熟练就开始学框架，等于地基没打就起高楼，你可能会快一步，但是迟早要遇到瓶颈，甚至摔跟头。

3. 那么何时开始学习框架？我不建议新手一开始就直接使用框架。

就好像一开始学习编程语言，大家都不推荐直接使用IDE，一定要用命令行自己编译运行几个文件之后，了解清楚了之后才可以使用IDE，要不然对于底层原理不了解，遇到问题没法自己手动排查。

4. 使用框架也是一样，如果不是自己写多了重复性的代码，就很难理解框架为什么要这么设计。如果不尝试几种不同的实现，就很难理解框架为了灵活性而做出的设计和扩展点。如果不写几十个权限检查语句，就很难理解AOP到底有什么好处。

5. 框架这么好，我该全部使用框架吗？首先只有在规模以上的程序中，才有应用框架的必要，一个简单的程序没必要使用框架，当然如果你很熟练，使用也无所谓。

6. 要学习一下框架的核心源代码，要为扩展框架做好准备，因为虽然框架基本上还算灵活，但是面对错综复杂的业务需求，永远不可能面面俱到，而你不了解框架的话，可能会给你实现业务需求造成麻烦。这也是有些人坚持使用Servlet+JSP原生开发，而不是用框架的理由。

7. 只要程序大了，归根究底还是要使用框架的，不是用别人写好的，就是自己写一套。这里我不建议自己写，不要重复造轮子，总有专业造轮子的。你草草写就的往往不如别人已经千锤百炼的代码。除非你是为了学习与研究的目的，自己写，那就是一件很好的事情。