25 阅读源码的正确姿势

更新时间：2019-12-13 15:32:55

> 人的差异在于业余时间。——爱因斯坦

1. 前言

Java 学习和进阶离不开阅读源码，但是很多人只知道阅读源码却不知道如何阅读源码更有效。

很多人面对源码无从下手，也有很多人阅读源码刚开始就陷入细节，看着看着就晕了，很难坚持下去。

也有很多人看了很多源码，最终都“忘了”，没留下什么印象。

我自己也遇到过类似的问题，通过探索和交流总结了一些经验，在此分享给大家。

2. 读源码究竟读什么？

很多人只是知道阅读源码是进阶的一个重要步骤，但是在阅读之前并不是很清楚到底要通过源码学到什么。

如果读源码之前想不清楚这件事，很容易“走马观花”，收获无多。

通过阅读源码可以学习到很多知识，如：编码规范，包括类、函数、属性的命名，注释的规范等；优秀程序员的编程思想；学习一些高级的编程技巧；某些功能或特性的核心原理；可以学习到一些好的设计原则、设计模式如何落地。

3. 阅读源码的思路

阅读源码的方法和心态很重要，很多人想一口气吃个大胖子，急于求成最后适得其反。

很多人急躁的心情是可以理解的，想早点攻克某个框架源码，但是大家可以回想一下打游戏的场景，想打好游戏，通常需要学习各种通关技巧，需要先 “打野”。

下面介绍几个阅读源码的思路。

3.1 从设计者的角度看源码

从设计者的角度看源码是最有效的方式。

源码也是人写出来的，源码的作者编写代码之前也是在头脑中思考过的。

源码，尤其是复杂源码，都是符合 “任务拆分” 的原则的，即一个大的功能分为几个核心的步骤，分别编写代码。

这也符合罗伯特□C□马丁（Robert Cecil Martin）所提出的面向对象五大基本原则之一的：单一职责原则。

> 单一职责原则：一个类或者模块应该有且只有一个改变的原因

因此我们学习源码要想好编写这个功能应该有哪些步骤，再去和源码对比。

这样才能验证自己思考问题的角度是否正确，是否有遗漏。

通过对比能够清楚地知道作者为什么要这么设计，作者的源码比自己所设想的好在哪里，这样才不容易遗忘。

这就像学生时代做数学题一样，很多人会发现如果我们不做题就直接看答案，我们会认为问题都很简单，自己都会。但是真正脱离答案去做题时，往往并不会做。这也像我们拿着复杂迷宫的答案图纸去看迷宫时，会认为迷宫并不难，但是没有提前看答案时，破解迷宫的难度是要大很多的。

下面举一个非常简单的例子：

在开发时，需要借助 okhttp 封装一个 HTTP 请求工具类，其中涉及到编写一个判断请求是否成功的函数。

正如很多人认为地那样，在封装地函数中直接判断响应码是否等于 200 即可。

```
public boolean isSuccessful(Integer code) {
    return 200 == code;
}
```

但是当我们去查看 `okhttp3.Response` 的 `isSuccessful` 的写法：

```
/**
 * Returns true if the code is in [200..300), which means the request was successfully received,
 * understood, and accepted.
 */
val isSuccessful: Boolean
  get() = code in 200..299
```

突然发现我们的想法不够严谨，响应码从 200 到 299 都应该算请求成功。

这样我们对源码的某个细节的印象就会非常深刻，更加清晰地了解到自己思路的不严谨性。

如果我们的“猜想”核心的步骤和最终和源码比对，如果和作者的逻辑非常一致时，我们就很开心，这也是看源码的乐趣之一。如果不一致，通过对比完善自己的思路。

通过这种方式去读源码能够不断纠正我们的思路，不断发现我们的问题，这是阅读源码非常重要的一个目的。

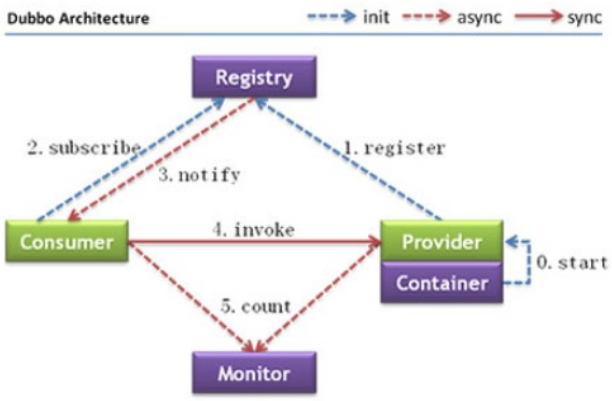
3.2 先整体后局部

俗话说 “磨刀不误砍柴工”，这几乎是尽人皆知的道理，但是学习编程时很多人依然会着急看源码，不重视背景知识，不重视框架的整体思想，导致后面浪费更多地时间。

为了避免过早陷于局部而缺乏全局观念，应该先从整体了解一个技术的核心模块再去学习每个具体模块的源码。

比如我们学习 dubbo 源码之前必须想了解该框架的主要模块以及之间的关系。

dubbo 架构图：



先了解架构的核心角色以及调用关系，再去学习源码会更容易一些。

先仔细阅读官方手册再去学习源码，很多人不重视官方手册，学习很久甚至工作很久，连核心技术栈的官方手册都没认真看过一遍，这是一件非常可怕的事情。

对要学习的技术有一个整体地了解之后，可以去拉取源码，去看源码包含哪些模块，每个模块的大体功能是什么，各个模块之间有什么关系等，然后再去看代码的细节。而大多数人读源码，会认为这些不重要，会急于读源码，导致效果不好。

3.3 由易到难

尤其是对于很多新手来说，连核心技术栈使用都不熟悉的情况下，直接看其源码很容易遭受很大打击。

因此要根据自己的阶段去选择适合自己的框架来阅读。

这一点和打游戏是非常一致的，一般开局都先“打野”，通过“打野”来提升等级获取装备等，再去和高级的敌人对抗。

对于初学者而言，可以先从开发中常用的简单的框架入手，如 commons-lang 、commons-collections、 guava 等。从看这些简单的源码积累经验，然后再去学习 spring 、spring boot、dubbo 等框架的源码。

另外要先保证能够熟练使用，再去学习源码效果会更好一些。如果连使用都不会就直接去学习源码，是一种非常不理智的行为。

另外学习从来不是匀速的，大家也明白“欲速则不达”的道理，建议可以先从简单的框架入手，积累经验后快速将这种学习的能力迁移到自己想研究的框架中去。

3.4 带着问题看源码

3.4.1 通用的问题

看源码和学某个技术之前，要重点思考几个能从整体理解该项目的问题：

- 这个项目主要核心功能是什么？
- 这个框架能解决什么问题？
- 有没有同类的框架，有啥异同？

很多人会忽略这些问题，认为这些问题不重要，导致虽然能用起来，却对框架的使用场景、解决的本质问题都不清楚。

3.4.2 工作中遇到的问题

对于很多人而言，大多数时间都花费在工作上，业余时间并没那么多，那么如何去学源码呢？

其实未必需要有大量完整地时间才可以去学习源码，**我们在开发过程中遇到问题时可以顺便进入源码来研究问题**。

在工作任务不是特别忙的时候，可以通过自己项目引用的 jar 包进入源码中看一下平时常用的注解是如何解析的，常用的函数具体实现是怎样的，常用的类中还有哪些其它函数等。

当学习和工作中遇到问题时，如果能够借机深挖，就可以借助这个问题带动源码的部分内容的学习。

如在 《虚拟机退出时机问题研究》小节所举的例子，下面代码打印语句还没来得及执行就结束了：

```
public class CompletableFutureDemo {
  public static void main(String[] args) {
    CompletableFuture.runAsync(() -> {
      try {
        TimeUnit.SECONDS.sleep(2L);
      } catch (InterruptedException ignore) {
      }
      System.out.println("异步任务");
    });
  }
}
```

最终我们通过进入 `runAsync` 函数的源码跟踪到 `ForkJoinPool#registerWorker` 函数发现， `ForkJoinPool` 的工作线程类型为守护者线程。

我们就借着这个机会，学习了 `CompletableFuture` 源码的部分知识点。

下面介绍另外一个问题，比如某同学使用 MyBatis 时，运行项目测试时发现找不到自定定义好的 Mapper，咋回事呢？

此时你要想出各种可能性：

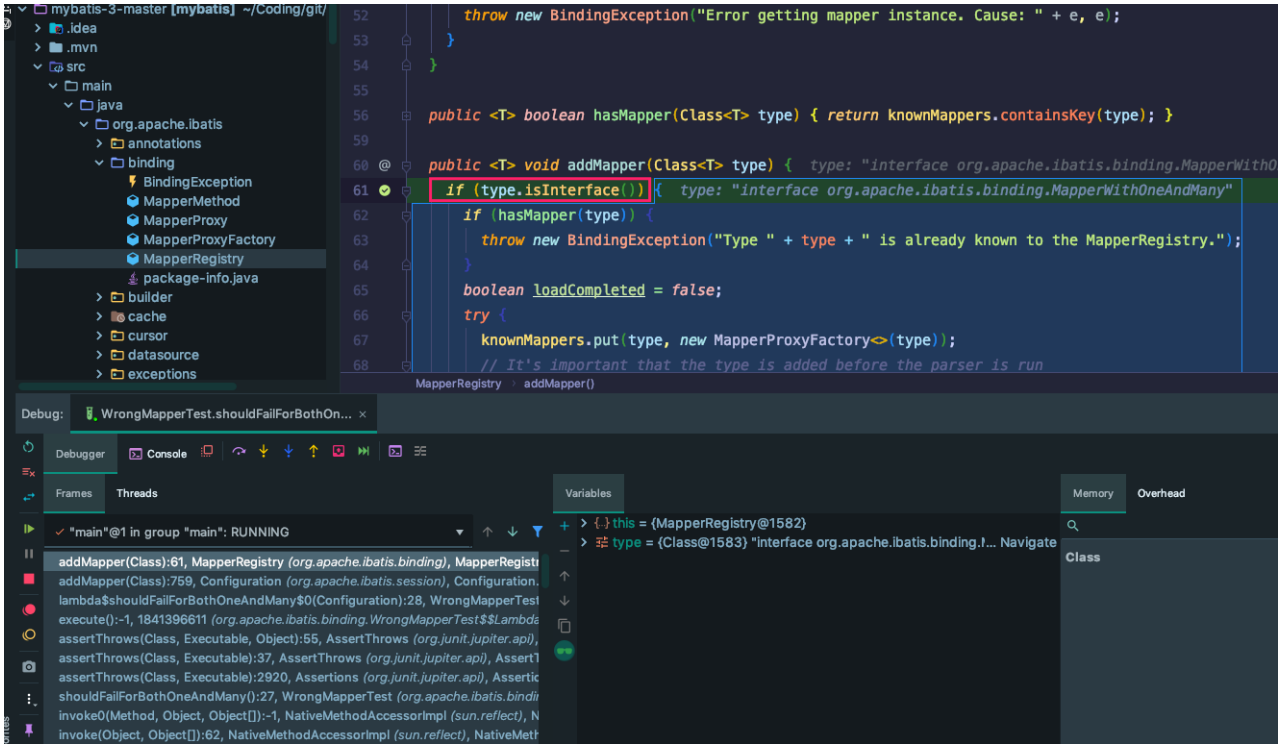
- 包名是不是写错了扫描不到？
- 接口是不是写成类了？

等等最有可能的问题，然后依次排查。

最终发现是因为自己误将 Mapper 接口定义为类导致的，将类 (class) 改成接口 (interface) 就好了。

那么为啥会这样呢？大家千万不要就此打住。

了解 MyBatis 会通过 MapperRegistry 来注册和获取 Mapper 对象的代理，我们进入添加 mapper 的核心代码：



可以看到该函数会先判断 Mapper 是否为接口类型，如果是接口类型才会注册此映射的代理对象，因此问题就非常明确了。

每一个问题都是我们学习源码的好机会，希望大家能够有这种意识。

很多人恰恰是平时不用心，临近找工作突击，才导致学啥都不深入，结果可想而知。

随着遇到的问题越来越多，看过常见的类的源码的函数越来越多，对源码的理解就越来越深刻，越来越熟悉。

3.4.3 看 issues

通过看开源项目的 issues， 你可以发现该项目的潜在 BUG。

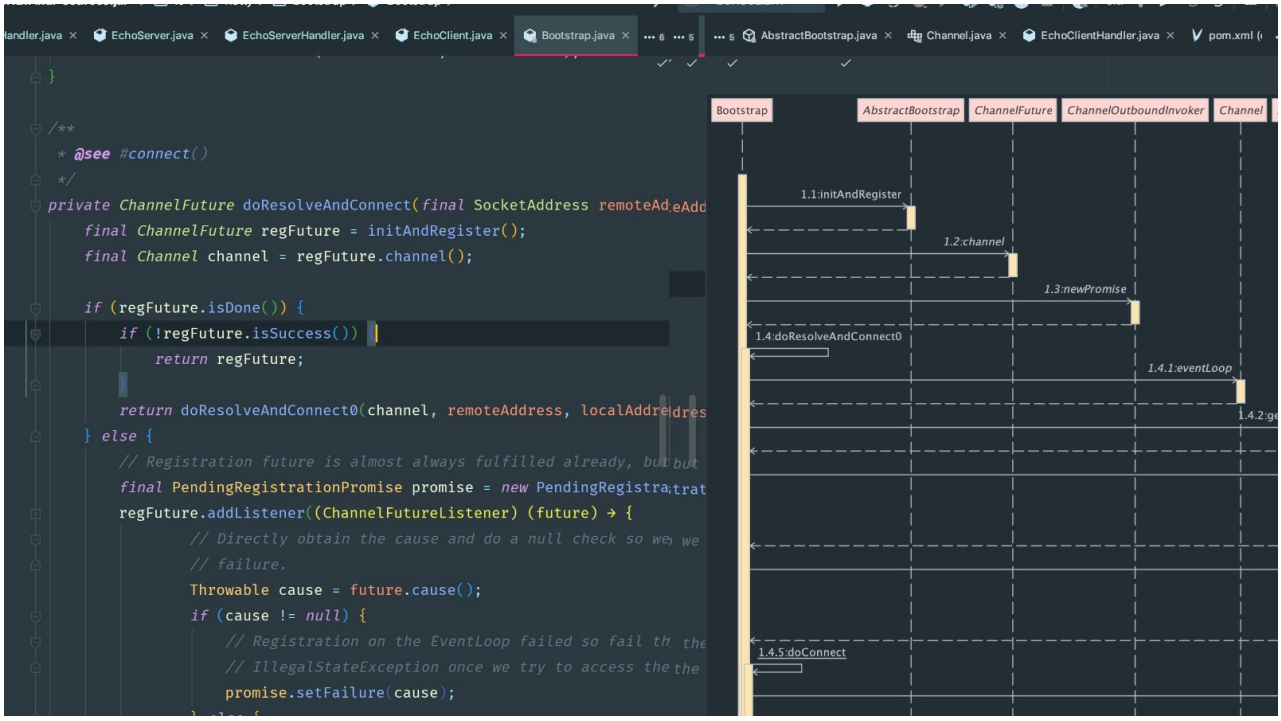
可以了解同一个问题，不同人的解决思路，以及官方最终采用的是哪种方案。

很多人的解决方案，对你实际的业务开发也有很大帮助。

3.4.4 看时序图

有些人可能会想到，可以根据源码画出时序图来理解源码，但是画图非常耗时，怎么办呢？

IDEA 插件 SequenceDiagram 就派上用场了，这个插件非常赞，可以根据源码绘制出调用时序图，对学习源码帮助极大。



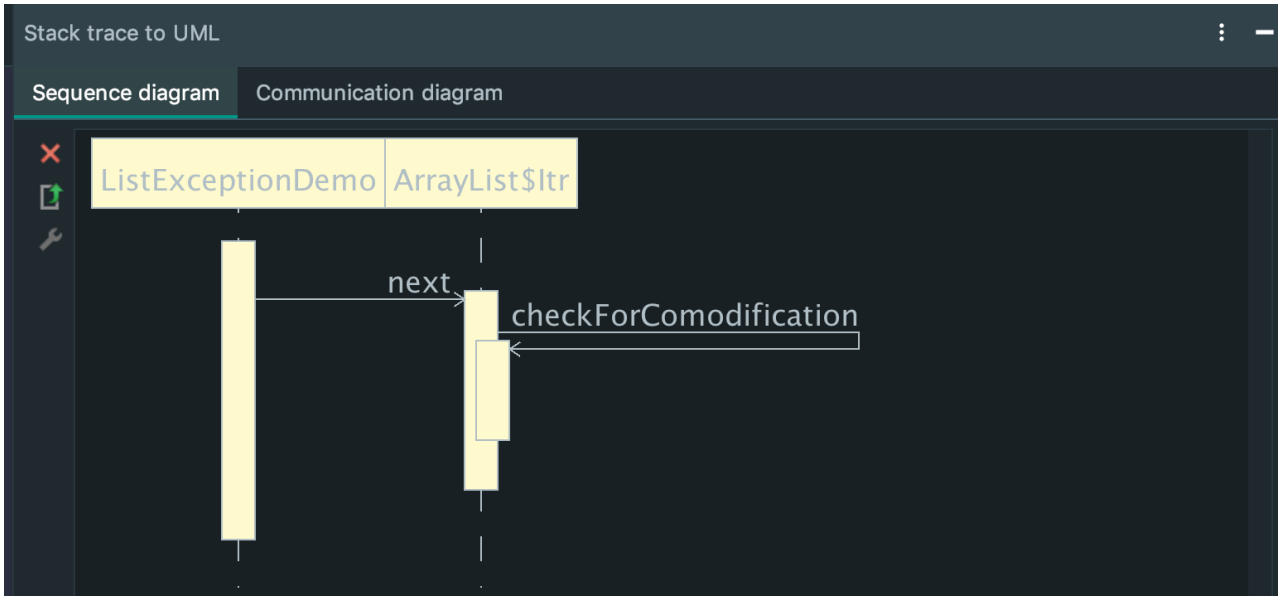
3.4.5 看错误堆栈信息

程序运行出错时，是我们学习的最佳时机。

大家可以通过 [Stack trace to UML 插件](Stack trace to UML) 绘制出出错的调用时序图，了解调用的顺序。

如下面出错信息：

安装好插件后，通过菜单：Analyze > Open Stack trace to UML plugin + Generate UML diagrams from stacktrace from debug ，将绘制出下面的时序图：



当异常堆栈信息非常多时，通过该插件绘制出的时序图将非常有助于帮助我们了解调用链，理解源码。

3.5 带着场景学源码

比如从设计模式的角度去学习源码。

可以从设计模式的六大原则来思考源码的设计，思考源码是如何体现这几种原则的。

> 设计模式六大原则：单一职责原则、里氏替换原则、依赖倒置原则、接口隔离原则、迪米特法则、开放封闭原则。

还可以结合《设计模式之禅》这本书或者菜鸟教程中设计模式的教程，了解具体某些设计模式的特点、使用场景、优点、缺点等。

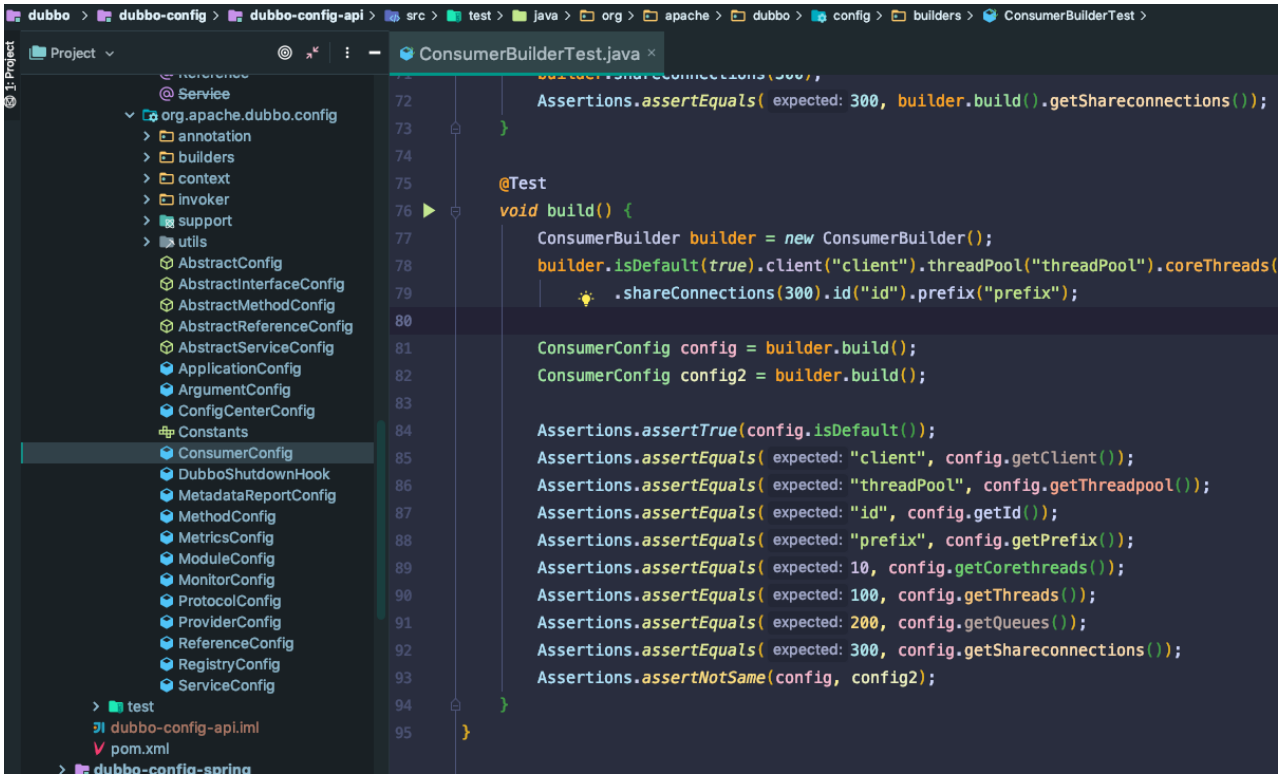
然后从 JDK 或者 Spring 等自己想学习的框架源码中去寻找这些设计模式的身影。

通过这种方式可以更清楚设计模式该如何落地，从更多角度去了解源码。

这里不详细展开，希望大家自行学习。

3.6 通过源码的单元测试来学习源码

正如前面一些章节所提到的，大多数知名的 Java 开源项目都会有非常完善的单元测试，这是我们学习源码的一个非常重要的突破口。我们可以运行单元测试来调试源码，熟悉核心类的功能。

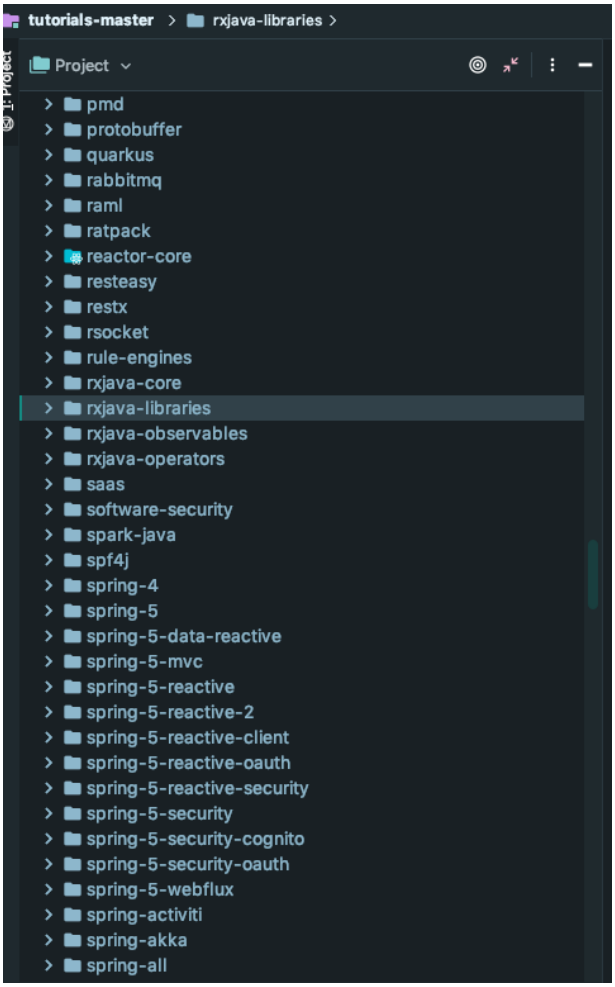


可以直接根据类名搜索，也可以通过找到该类，使用 “find usages” 功能来找到其单元测试代码。

3.7 通过 DEMO 学源码

大家可以使用官方的例子或者自己写例子运行，来体会某个项目的用法，研究其特性。

在这里推荐一个高质量的英文技术文章网站 baeldung, 几乎所有的文章都有 配套代码 , 我们可以直接通过该网站的代码运行学习某些知识点，某些框架。



大家学习某个框架，还可以自行去 github 找到相关的范例，运行学习。

另外，超级推荐大家通过自己开发的项目来学习 Spring 源码。大家可以对照着官方文档、对照着 Spring 的源码教程等，观察自己项目中某个 Spring 类的使用，还可以在项目测试时偶尔进到源码中断点，通过调试自己的项目来学习源码。

4. 阅读源码的技巧

4.1 实现“简易版”是学习的重要途径

比如学习 Spring 源码之前，可以根据自己平时使用 Spring 的方式，自己实现简易版的 Spring，记录自己编写代码的核心步骤，以及核心步骤的缺点和遇到的问题。待真正去阅读源码时，很多问题豁然开朗。

可能很多人会认为，不是所有的代码都有简易版。的确如此，但是只要思维灵活，方法总比困难多。

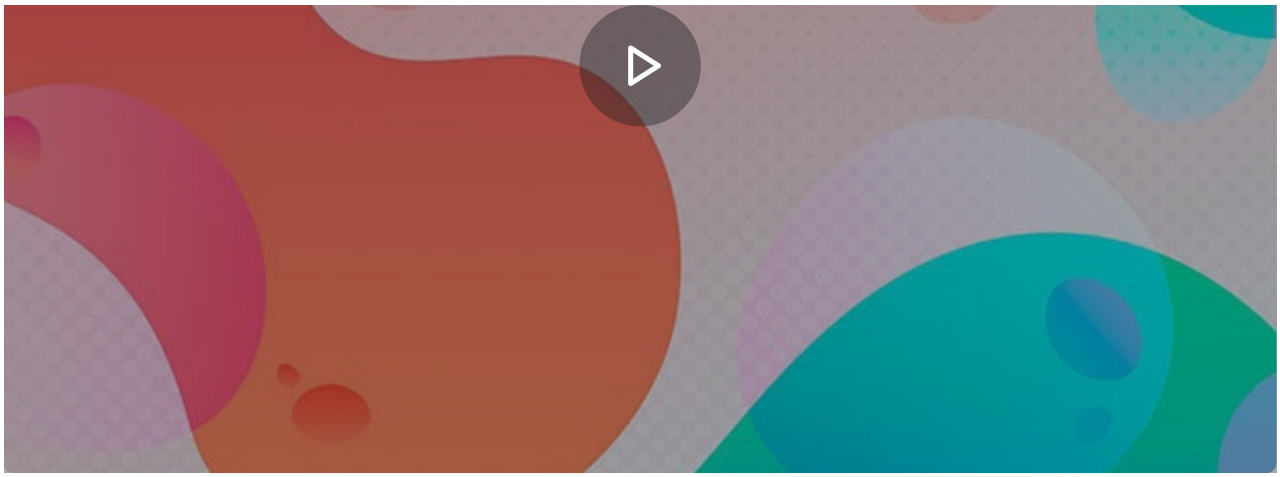
如可以购买或者寻找《Spring5 核心原理与 30 个类手写实战》书本所配套的简单版 Spring 代码，并且自己尝试从最简单版去改编，梳理清楚核心逻辑，并记录这个过程中遇到的困难。再去读 Spring 源码就会容易很多。

比如想读 dubbo 源码，可以在 github 上找一些简单的 Java RPC 框架看会后再去看 dubbo 的源码。

4.2 寻找程序入口是一个学习源码的切入点

通过寻找程序启动的入口，对入口断点调试，可以从源头了解框架的启动流程和运行原理。

可以通过打断点，然后通过调用栈逆向寻找入口；可以找网上的博客的源码分析找到入口打断点。



4.3 阅读源码时要重视函数的命名

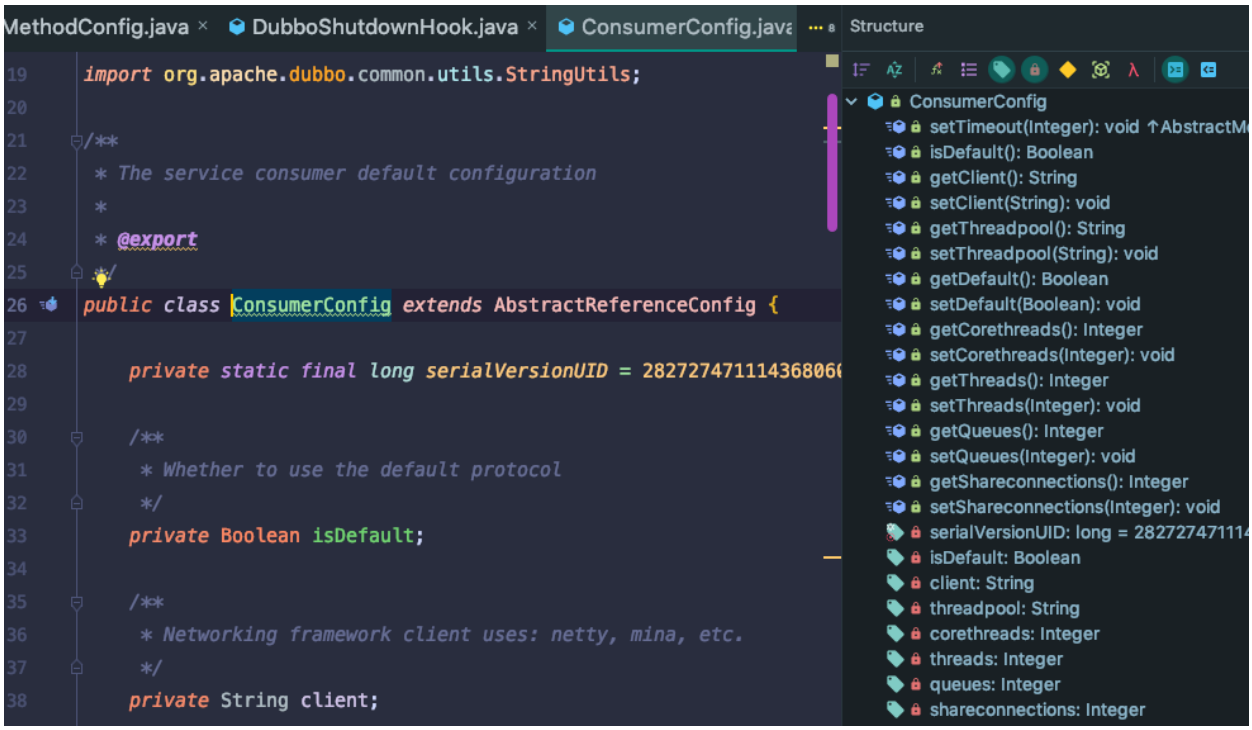
往往优秀的源码函数命名都非常贴切。可以通过 IDEA 的 structure，来了解源码中某个类的核心函数。

核心类有哪些核心的函数，这些函数的功能又是什么，对学习源码帮助很大。

通过单个函数快速了解其意图，对学习源码帮助很大。

4.4 多看函数列表

在看源码时建议打开函数列表，进入某个类时优先看该类有哪些公有函数。



这样做有助于帮助你从整体了解该类，更全面地了解一个类的功能。

4.5 阅读源码时要重视源码的注释

优秀的开源项目的类、函数甚至成员变量都会有非常详尽的注释。

注释可以快速帮助我们理解源码，帮助我们了解一些重要细节。

比如很多代码会给出其核心步骤，此时一定要先阅读函数上面的注释和内部给出的核心步骤再去读源码。

比较典型的一个案例 `java.util.concurrent.ThreadPoolExecutor#execute` ：

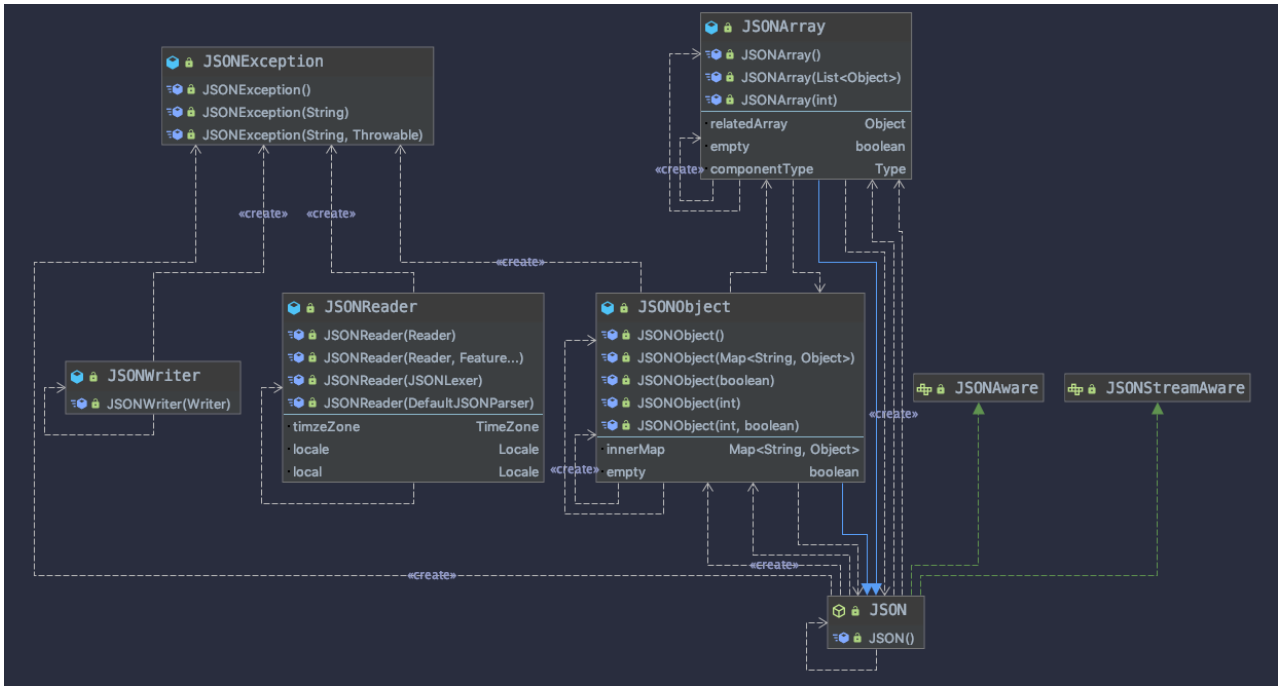
```
/**
 * Executes the given task sometime in the future.  The task
 * may execute in a new thread or in an existing pooled thread.
 *
 * If the task cannot be submitted for execution, either because this
 * executor has been shutdown or because its capacity has been reached,
 * the task is handled by the current {@code RejectedExecutionHandler}.
 *
 * @param command the task to execute
 * @throws RejectedExecutionException at discretion of
 *         {@code RejectedExecutionHandler}, if the task
 *         cannot be accepted for execution
 * @throws NullPointerException if {@code command} is null
 */
public void execute(Runnable command) {
    if (command == null)
        throw new NullPointerException();
    /*
     * Proceed in 3 steps:
     *
     * 1. If fewer than corePoolSize threads are running, try to
     * start a new thread with the given command as its first
     * task.  The call to addWorker atomically checks runState and
     * workerCount, and so prevents false alarms that would add
     * threads when it shouldn't, by returning false.
     *
     * 2. If a task can be successfully queued, then we still need
     * to double-check whether we should have added a thread
     * (because existing ones died since last checking) or that
     * the pool shut down since entry into this method. So we
     * recheck state and if necessary roll back the enqueuing if
     * stopped, or start a new thread if there are none.
     *
     * 3. If we cannot queue task, then we try to add a new
     * thread.  If it fails, we know we are shut down or saturated
     * and so reject the task.
     */
    int c = ctl.get();
    if (workerCountOf(c) < corePoolSize) {
        if (addWorker(command, true))
            return;
        c = ctl.get();
    }
    if (isRunning(c) && workQueue.offer(command)) {
        int recheck = ctl.get();
        if (! isRunning(recheck) && remove(command))
            reject(command);
        else if (workerCountOf(recheck) == 0)
            addWorker(null, false);
    }
    else if (!addWorker(command, false))
        reject(command);
}
```

4.6 关注目标类继承的类或者实现的接口

目标类的父类和实现的接口是研究该类功能和特征的重要突破口。

借助前面章节讲到的调试技巧，查看调用栈，运行表达式等可以极大地帮助我们理解源码。通过 IDEA 提供的类图功能，可以帮助我们理解不同类之间的关系。

如下图所示，通过 IDEA 自带的类图工具绘制出 fastjson 核心类之的类图，通过类图的选项来控制显示的内容和可见性。



4.7 其它

大家可以使用前面调试章节所学到的查看调用栈、设置条件断点、查看加载的对象等调试功能来帮助大家学习源码。

大家还可以跟着某个框架的专栏作者的思路去深入学习某个具体框架的源码。

5. 总结

本节主要讲述如何阅读源码，讲到了阅读源码的思路和一些技巧。希望通过本文的介绍大家可以更高效地阅读源码，提高进阶的速度。

下一节将讲述重构的相关知识。

参考资料

}

24 代码调试的正确姿势

26 代码重构的正确姿势