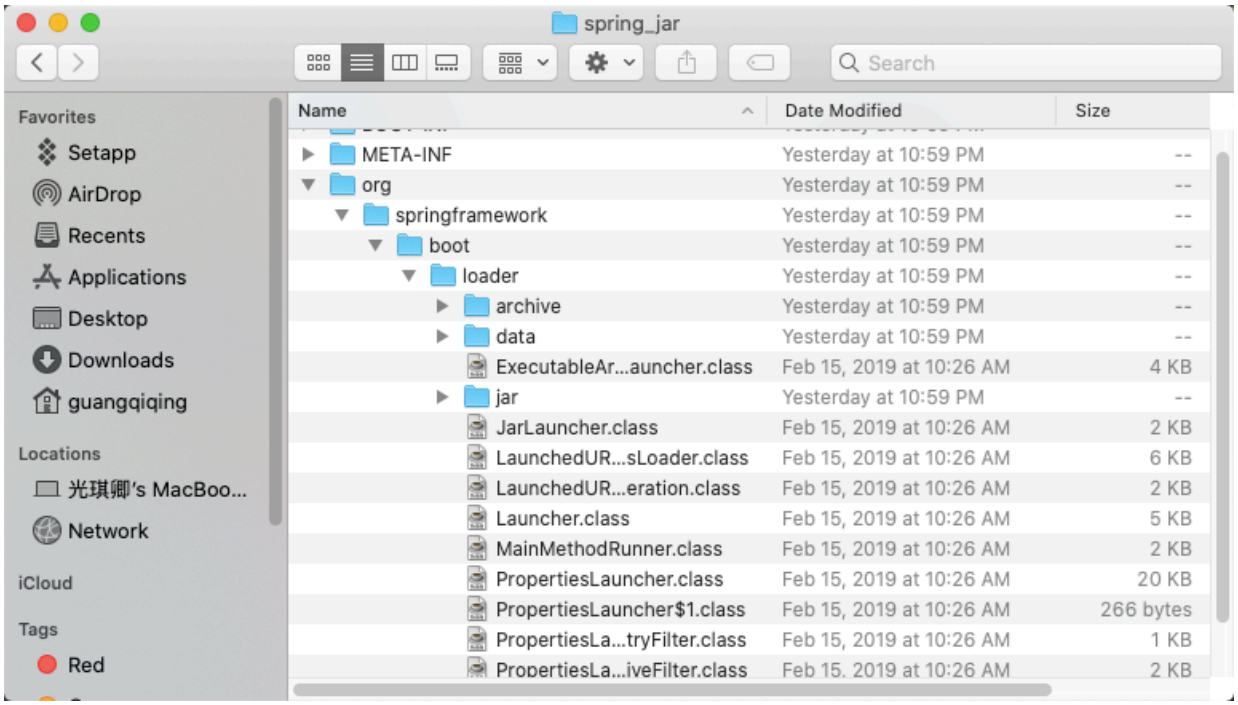
day1 spring classLoader

```
unzip jarName -d dir
```




```
BOOT-INF
META-INF
org
    springframe
        JarLauncher.class
```

JArLauncher

```
/**
 * {@link Launcher} for JAR based archives. This launcher assumes that dependency jars
    are
 * included inside a {@code /BOOT-INF/lib} directory and that application classes are
 * included inside a {@code /BOOT-INF/classes} directory.
 *
 * @author Phillip Webb
 * @author Andy Wilkinson
 */
```

```
public class JarLauncher extends ExecutableArchiveLauncher {

    static final String BOOT_INF_CLASSES = "BOOT-INF/classes/";

    static final String BOOT_INF_LIB = "BOOT-INF/lib/";

    public JarLauncher() {
    }

    protected JarLauncher(Archive archive) {
        super(archive);
    }

    @Override
    protected boolean isNestedArchive(Archive.Entry entry) {
        if (entry.isDirectory()) {
            return entry.getName().equals(BOOT_INF_CLASSES);
        }
        return entry.getName().startsWith(BOOT_INF_LIB);
    }

    public static void main(String[] args) throws Exception {
        new JarLauncher().launch(args);
    }

}
```

fatJar

为什么spring不遵守jar的普通规定, 还能直接通过jar运行.
jar目录顶层必须有main方法, jar里面不能有jar, 只能有class.

spring打包的jar下面org里面有main方法 里面调用了main的class loader去加载.

好处: 1. 防止第三方依赖与本地混乱 2. 防止第三方依赖重名

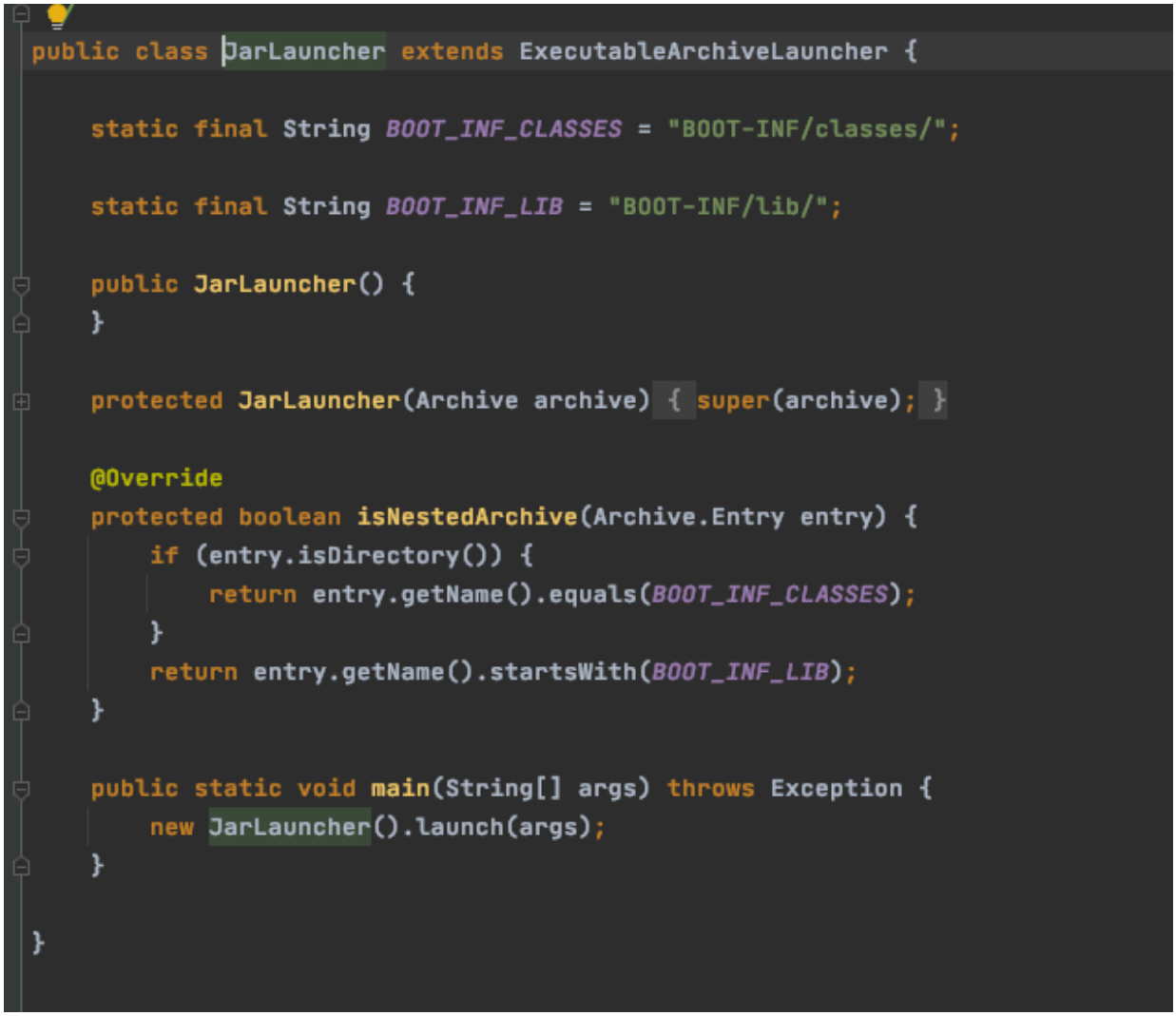
应用类加载.

```
Manifest-Version: 1.0
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: com.shengsiyuan.boot.MyApplication
```

link:

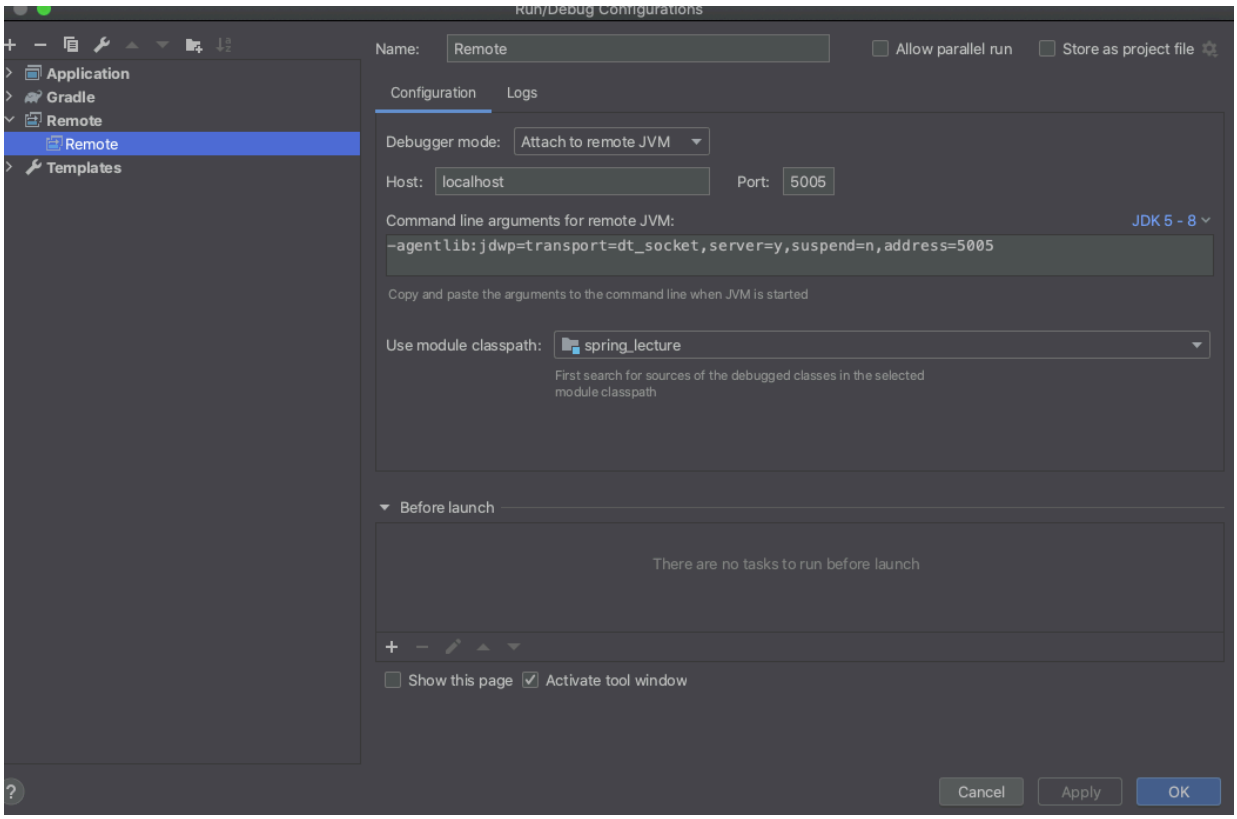
线程上下文类加载器（Context ClassLoader）：线程上下文类加载器是从JDK1.2开始引入的，类Thread中的getContextClassLoader()与setContextClassLoader(ClassLoader cl)分别用来获取和设置上下文类加载器。如果没有通过setContextClassLoader(ClassLoader cl)进行设置的话，线程将继承其父线程的上下文类加载器。Java应用运行时初始线程的上下文类加载器是系统类加载器【所以这也是为啥上面的第一行输出是系统类加载器的原因之所在】。在线程中运行的代码可以通过该类加载器来加载类与资源。

可见都是抽象接口，而且是位于rt.jar中，其实现肯定是由不同的数据库厂商来实现，那么问题就来了：这些标准都是由根类加载器所加载的，但是具体的实现是由具体的厂商来做的，那肯定是需要将厂商的jar放到工程的classpath当中来进行使用，很显然厂商的这些类是没办法由启动类加载器去加载，会由应用类加载器去加载，而根据"父类加载器所加载的类或接口是看不到子类加载器所加载的类或接口，而子类加载器所加载的类或接口是能够看到父类加载器加载的类或接口的"这一原则，那么会导致这样一个局面：比如说java.sql包下面的某个类会由启动类加载器去加载，该类有可能会要访问具体的实现类，但具体实现类是由应用类加载器所加载的，java.sql类加载器是根据看不到具体实现类加载器所加载的类的，这就是基于双亲委托模型所出现的一个非常致命的问题，这种问题不仅是在JDBC中会出现，在JNDI、xml解析等SPI（Service Provider Interface）场景下都会出现的，所以这里总结一下：父ClassLoader可以使用当前线程Thread.currentThread().getContextLoader()所指定的ClassLoader加载的类，这就改变了父ClassLoader不能使用子ClassLoader或者其它没有直接父子关系的ClassLoader加载的类的情况，既改变了双亲委托模型。线程上下文类加载器就是当前线程的Current ClassLoader。在双亲委托模型下，类加载是由下至上的，既下层的类加载器会委托上层进行加载。但是对于SPI来说，有些接口是Java核心库所提供的，而Java核心库是由启动类加载器来加载的，而这些接口的实现去来自于不同的jar包（厂商提供）。Java的启动类加载器是不会加载其它来源的jar包，这样传统的双亲委托模型就无法满足SPI的要求。而通过给当前线程设置上下文类加载器，就可以由设置的上下文类加载器来实现对于接口实现类的加载。



```java
public class JarLauncher extends ExecutableArchiveLauncher {

    static final String BOOT_INF_CLASSES = "BOOT-INF/classes/";

    static final String BOOT_INF_LIB = "BOOT-INF/lib/";

    public JarLauncher() {
    }

    protected JarLauncher(Archive archive) { super(archive); }

    @Override
    protected boolean isNestedArchive(Archive.Entry entry) {
        if (entry.isDirectory()) {
            return entry.getName().equals(BOOT_INF_CLASSES);
        }
        return entry.getName().startsWith(BOOT_INF_LIB);
    }

    public static void main(String[] args) throws Exception {
        new JarLauncher().launch(args);
    }

}
```

远程调试 intellej 里面设置debug




```
java -agentlib:jdwp=transport=dt_socket, server=y,suspend=y,address=5050 -jar
    spring_lecture-1.0.jar
```