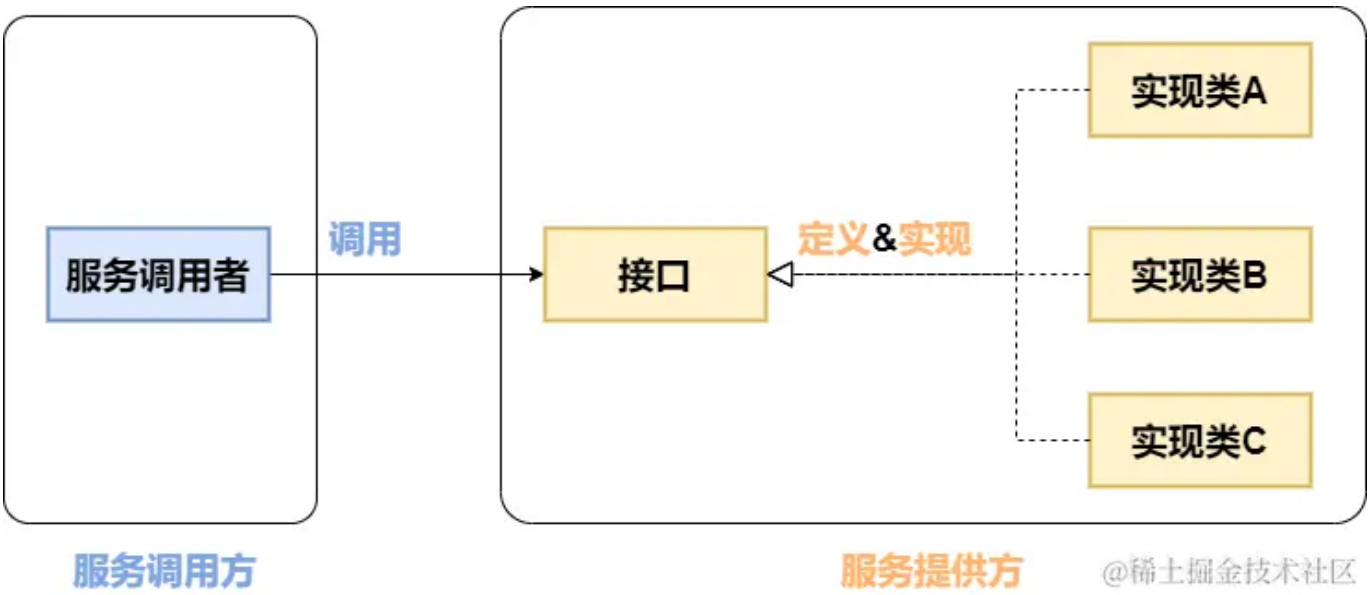
Java的SPI机制

概念

API

API在我们日常开发工作中是比较直观可以看到的，比如在 Spring 项目中，我们通常习惯在写 service 层代码前，添加一个接口层，对于 service 的调用一般也都是基于接口操作，通过依赖注入，可以使用接口实现类的实例。

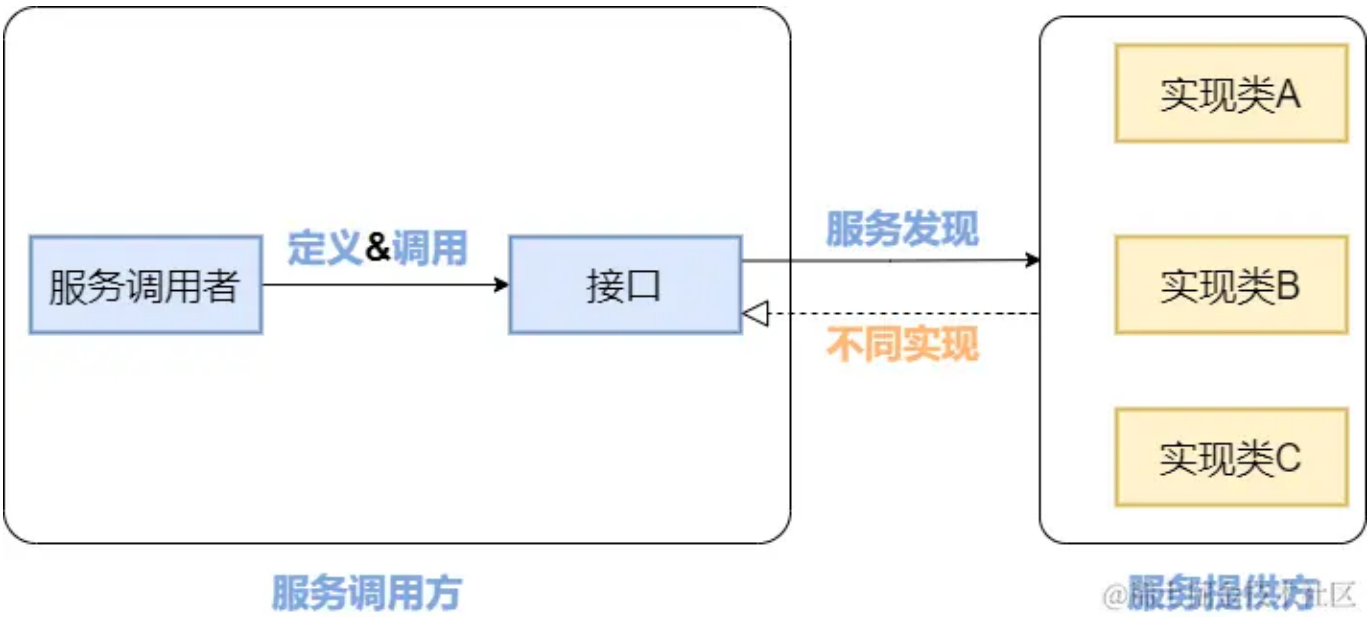


如上图所示，服务调用方无需关心接口的定义与实现，只进行调用即可，**接口、实现类都是由服务提供方提供**。服务提供方提供的接口与其实现方法就可称为**API**，API中所定义的接口无论是在概念上还是具体实现，都更接近服务提供方，通常接口与实现类在同一包中；

SPI

如果我们将接口的定义放在调用方，服务的调用方定义一个接口规范，可以由不同的服务提供者实现。并且，调用方能够通过某种机制来发现服务提供方，通过调用接口使用服务提供方提供的功能，这就是SPI的思想。

SPI 的全称是Service Provider Interface，字面意思就是**服务提供者的接口**，是由服务提供者定义的接口。



SPI与API详细对比

对比维度	SPI（服务提供者接口）	API（应用程序接口）
定义	框架定义接口，服务提供者实现接口	框架定义接口，调用者直接使用接口
使用方式	通过`ServiceLoader`动态加载实现类	直接调用接口或具体实现类
设计目的	实现扩展机制，允许第三方插件式扩展	提供标准功能接口，隐藏实现细节
耦合度	松耦合（接口与实现分离）	紧耦合（调用者依赖具体API）
实现控制	由服务提供者控制实现逻辑	由框架开发者控制实现逻辑
扩展性	高扩展性（动态添加新实现无需修改代码）	低扩展性（需修改代码才能扩展功能）
发现机制	通过`META-INF/services`配置文件自动发现服务实现	需显式调用具体类或方法
典型应用场景	JDBC驱动加载、SLF4J日志绑定、Spring Boot自动配置	Java集合框架、网络编程API、IO库
加载时机	延迟加载（首次调用时加载）	编译时或启动时加载
配置文件	必须存在`META-INF/services/接口全限定名`文件	无需配置文件
错误处理	加载失败抛出`ServiceConfigurationError`	直接抛出编译错误或运行时异常
模块化支持	需在`module-info.java`中添加`provides`/`with`语句	需在`module-info.java`中添加`exports`语句
性能影响	存在类加载和实例化开销	直接调用，性能更高
版本兼容性	实现类版本需兼容接口定义	接口变更会导致调用方代码不兼容
控制反转（IoC）	框架控制实现的选择（好莱坞原则：*Don't call us*）	调用方控制实现的选择
代码侵入性	无侵入（通过配置扩展）	需显式调用具体类（存在侵入性）

核心区别总结

1. **方向性**
    - **API**：面向调用者，定义**如何调用**功能（由框架→调用者）
    - **SPI**：面向实现者，定义**如何扩展**功能（由框架←实现者）
2. **设计哲学**
    - **API**：提供标准化的功能入口
    - **SPI**：提供可插拔的扩展机制
3. **技术实现**
    - **API**：通过接口/抽象类直接暴露功能
    - **SPI**：通过`ServiceLoader`+配置文件动态发现实现
4. **典型示例**
    - **SPI**：JDBC的`DriverManager`加载不同数据库驱动
    - **API**：`List`接口定义集合操作规范

核心源码

一. ServiceLoader 类定义

```
public final class ServiceLoader<S> implements Iterable<S> {
    // 配置文件前缀
    private static final String PREFIX = "META-INF/services/";

    // 服务接口的 Class 对象
    private final Class<S> service;

    // 类加载器
    private final ClassLoader loader;

    // 已加载的服务缓存
    private LinkedHashMap<String,S> providers = new LinkedHashMap<>();

    // 延迟迭代器
    private LazyIterator lookupIterator;

    // 访问控制上下文
    private final AccessControlContext acc;
}
```

二、核心加载流程源码分析

1. 初始化阶段 load() 方法

```
public static <S> ServiceLoader<S> load(Class<S> service) {
    // 关键点：使用当前线程的上下文类加载器
    ClassLoader cl = Thread.currentThread().getContextClassLoader();
    return new ServiceLoader<>(service, cl);
}

private ServiceLoader(Class<S> svc, ClassLoader cl) {
    service = Objects.requireNonNull(svc, "Service interface cannot be null");
    loader = (cl == null) ? ClassLoader.getSystemClassLoader() : cl;
    acc = (System.getSecurityManager() != null) ? AccessController.getContext() :
null;
    reload();
}

public void reload() {
    providers.clear();
    lookupIterator = new LazyIterator(service, loader); // 创建延迟迭代器
}
```

关键设计：

- 使用线程上下文类加载器（TCCL）**解决类加载器委派问题**
- 初始化时仅创建迭代器，不立即加载实现类

2. 延迟加载实现（LazyIterator）

```
private class LazyIterator implements Iterator<S> {
    // 配置文件枚举器
    Enumeration<URL> configs;
    // 待解析的类名集合
    Iterator<String> pending;
    // 下一个服务实现类名
    String nextName;

    public boolean hasNext() {
        if (nextName != null) return true;
        if (configs == null) {
            try {
                // 关键点：拼接配置文件路径
                String fullName = PREFIX + service.getName();
                configs = loader.getResources(fullName);
            } catch (IOException x) { /*...*/ }
        }
        while ((pending == null) || !pending.hasNext()) {
            if (!configs.hasMoreElements()) return false;
            // 关键点：解析配置文件
            pending = parse(service, configs.nextElement());
        }
        nextName = pending.next();
        return true;
    }

    public S next() {
        if (!hasNext()) throw new NoSuchElementException();
        String cn = nextName;
        nextName = null;
        try {
            // 关键点：类加载与实例化
            Class<?> c = Class.forName(cn, false, loader);
            if (!service.isAssignableFrom(c)) {
                throw new ClassCastException();
            }
            S p = service.cast(c.getConstructor().newInstance());
            providers.put(cn, p); // 存入缓存
            return p;
        } catch (Throwable x) { /*...*/ }
    }
}
```

关键流程：

1. 动态拼接 `META-INF/services/接口全名` 路径
2. 通过 `ClassLoader.getResources()` 获取所有同名配置文件

3. 合并所有配置文件的类名条目

4. 按需执行类加载和实例化

```
1. ServiceLoader.load() 初始化
   └── 创建 LazyIterator
       └── 收集所有 META-INF/services/接口名 资源文件

2. 迭代时触发加载
   └── hasNext() 解析配置文件
       └── next() 实例化实现类
           ├── Class.forName() 加载类
           ├── 验证接口类型
           └── 反射创建实例
```

三 、实践

1. 实现类缓存优化

```
// 自定义带缓存的 ServiceLoader
public class CachedServiceLoader<S> {
    private static final Map<Class<?>, List<?>> cache = new ConcurrentHashMap<>();

    public static <S> List<S> loadAll(Class<S> service) {
        return (List<S>) cache.computeIfAbsent(service,
            key -> StreamSupport.stream(
                ServiceLoader.load(service).spliterator(), false)
                .collect(Collectors.toList()));
    }
}
```

2. 实现类过滤

```
public static <S> Optional<S> findFirst(Class<S> service, Predicate<S> predicate) {
    ServiceLoader<S> loader = ServiceLoader.load(service);
    for (S provider : loader) {
        if (predicate.test(provider)) {
            return Optional.of(provider);
        }
    }
    return Optional.empty();
}
```

应用

1. JDBC 驱动加载（`java.sql.Driver`）
2. SLF4J 日志门面实现绑定
3. Dubbo 扩展点机制

SPI 优劣分析

优势：

- 实现组件化架构
- 符合开闭原则（对扩展开放，修改关闭）
- 框架与实现解耦

局限：

- 无法按需加载（全量加载所有实现）
- 不支持参数化构造函数
- 缺乏完善的依赖注入机制

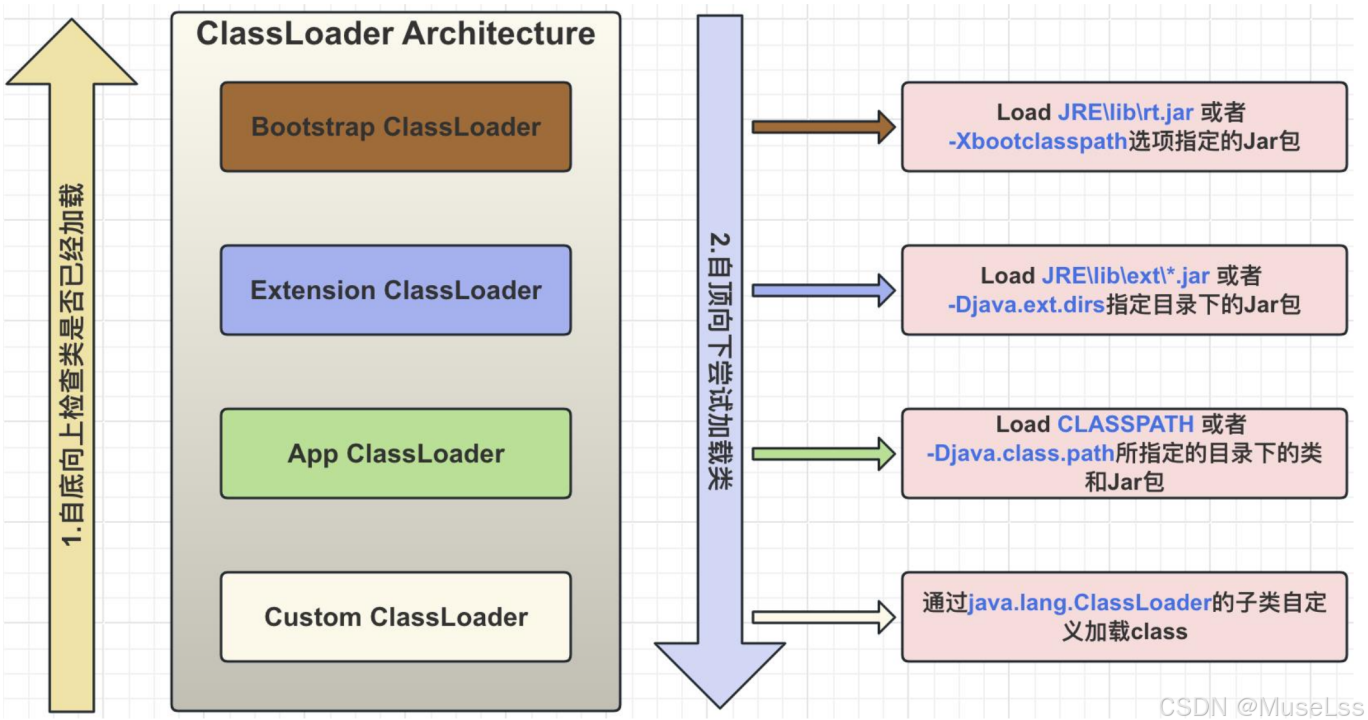
双亲委派机制&SPI

核心定义

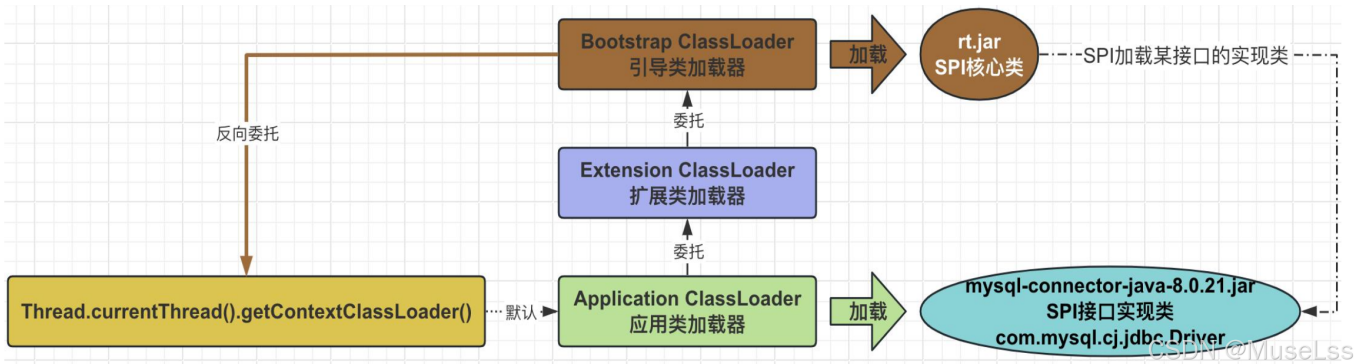
双亲委派是 Java 类加载器（ClassLoader）的工作机制，指当一个类加载器收到加载请求时：

1. **优先委派给父类加载器**尝试加载
2. **父加载器无法完成时**才由自己加载
3. 最终到达最顶层的启动类加载器（Bootstrap ClassLoader）

这种层级递进的加载模式形成了类加载器的树状结构体系。



为什么说SPI打破了双亲委派模式



如何“破坏”?

SPI核心类在rt.jar 位于Bootstrap ClassLoader启动类加载器中，但是SPI需要实现来自不同厂商提供的数据库Driver实现类，而这些来自不同厂商的SPI接口实现类（如com.mysql.cj.jdbc.Driver）位于Application ClassLoader应用类加载器中

为何说“破坏”?

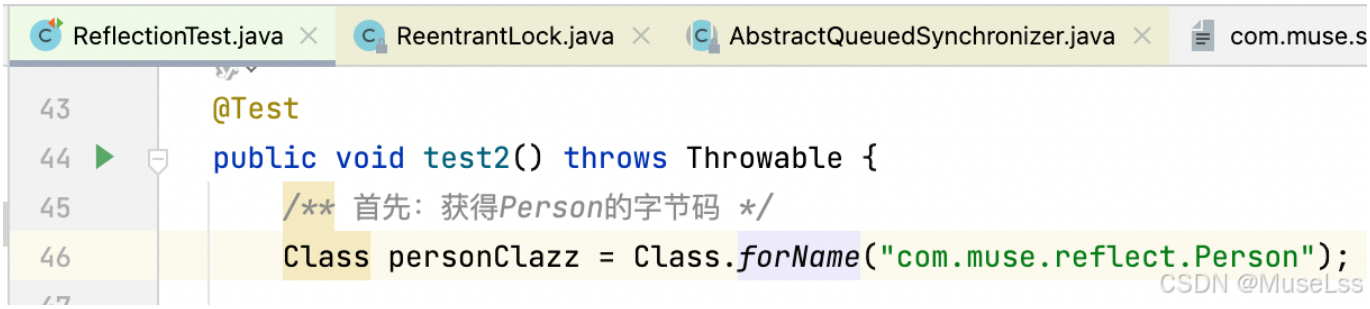
尽管SPI本身是由启动类加载器加载，但是它间接的通过应用类加载器加载第三方驱动类，绕过了严格的双亲委派机制

为什么要破坏?

因为使用双亲委派有一定的局限性，在正常情况下，用户代码一定是依赖核心类库的，所以双亲委派加载机制是没有问题的，但是在加载核心类库时，又要反过来使用用户代码，双亲委派就无法满足。这是什么样的一个场景呢?

比如jdbc利用Driver.Manager.getConnection获取连接时，DriverManager是由根类加载器Bootstrap ClassLoader加载的，在加载DriverManager时，会执行其静态方法，加载驱动程序（也就是Driver接口的实现类），这些实现类都是由第三方数据库厂商提供，根据双亲委派原则，第三方类不可能被根类加载器加载

双亲委派机制源码




```
public static Class<?> forName( @NonNls String className)
            throws ClassNotFoundException {
    Class<?> caller = Reflection.getCallerClass();
    return forName0(className,  initialize: true, ClassLoader.getClassLoader(caller), caller);
}
```


这里forName进行加载类，forName方法的getClassLoader是获取caller对应的ClassLoader,caller就是指当前调用的forName方法的那个类，即ReflectionTest这个类，因为这个类是自己写的，所以在CLASSPATH路径下，获取到的加载器应

假设这里按照双亲委派的加载思想，这里ServiceLoader是属于rt包，属于根加载器，根加载器去加载，肯定是找不到的，而且你也没有向下去委托扩展类加载器的路，因为你**起步就是根加载器，没法往下递归了，根加载器加载不到，就结束了**

但是，SPI这里并没有这样去做，而是从Thread.currentThread().getContextClassLoader()获取类加载器，**这里获取到的是应用类加载器**，然后由应用类加载器完成加载