



SOFASTACK

模块化方案演进思考

黄挺（鲁直）@ 蚂蚁金服



SOFASTACK

SOFA 是什么



SOFA --- Scalable Open Financial Architecture

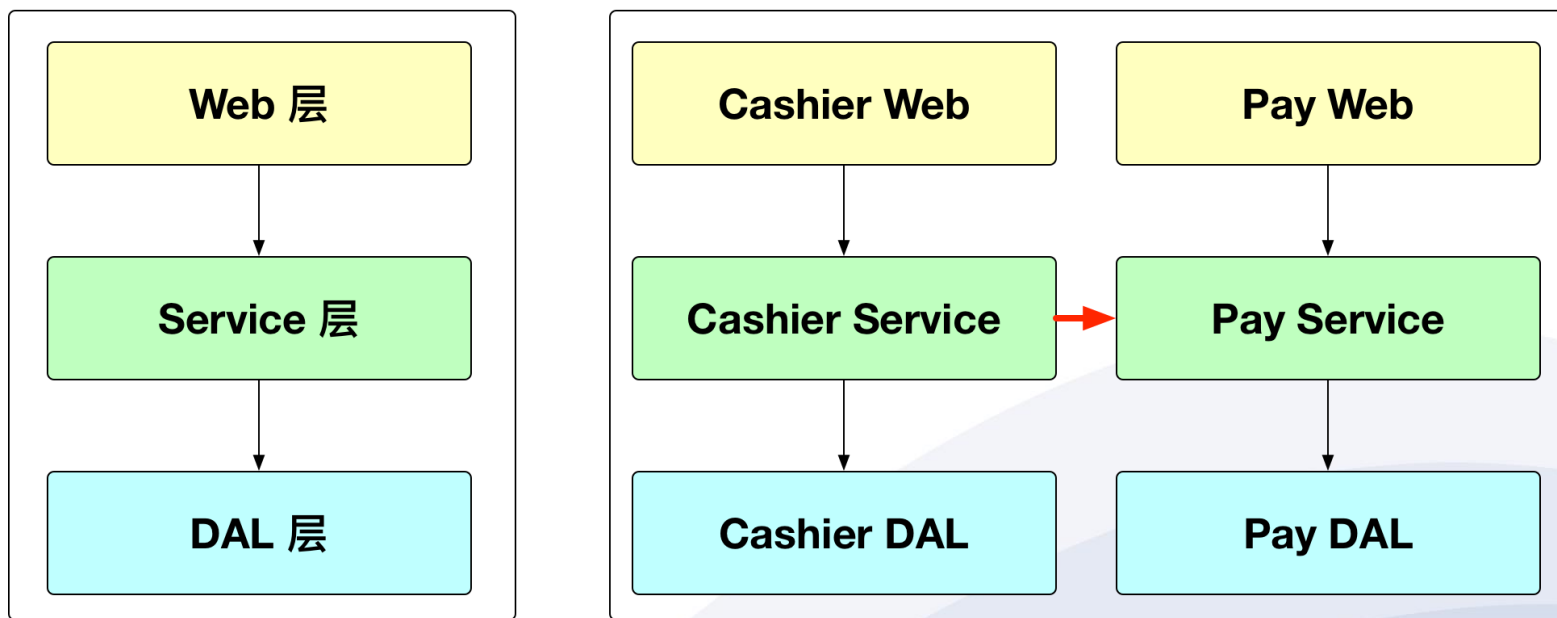
SOFA 中间件是蚂蚁金服开源的金融级分布式中间件



SOFASTACK

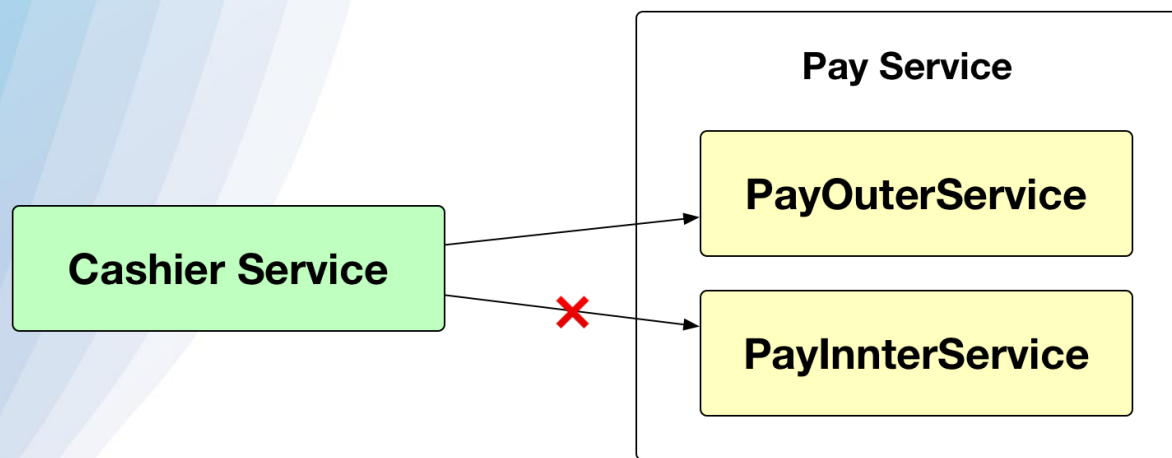


常见的模块化方式



常见的一个工程的模块的划分方式

常见的模块化的问题

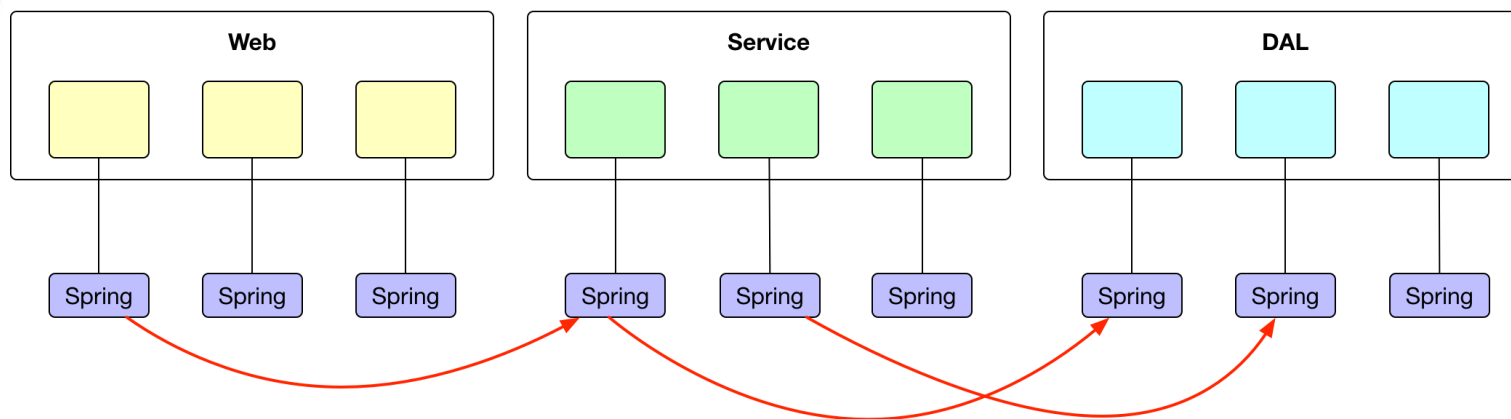


- 不合理的模块化方案会导致应用模块的高度耦合
- 不合理的模块化方案会导致研发成本的上升

调用了别的模块里面的内部服务/类:

```
public class CashierService {  
    @Autowired  
    PayInnerService payInnerService;  
  
    public void doSomething() {  
    }  
}
```

SOFA 模块化



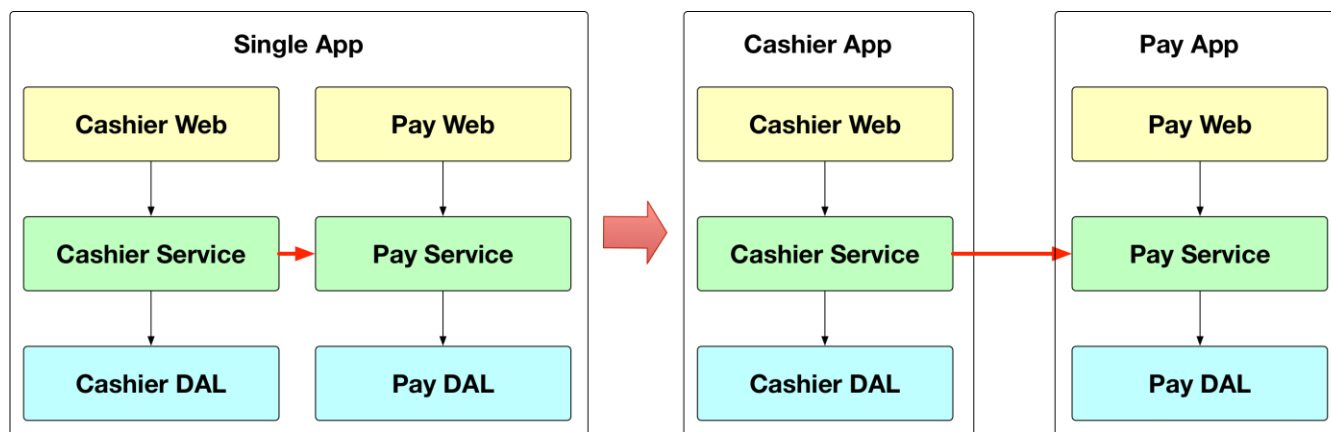
服务发布:

```
<sofa:service interface="com.alipay.demo.SampleService" ref="sampleService"/>
```

服务引用:

```
<sofa:reference interface="com.alipay.demo.SampleService" id="sampleService"/>
```


模块化 – 快速拆分服务



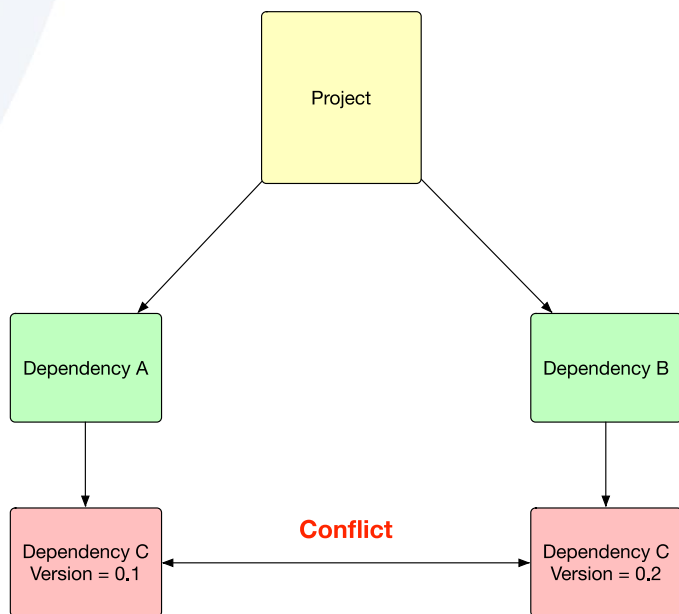
服务发布:

```
<sofa:service interface="com.alipay.demo.SampleService" ref="sampleService"/>
<sofa:service interface="com.alipay.demo.SampleService" ref="sampleService">
  <sofa:binding.bolt/>
</sofa:service>
```

服务引用:

```
<sofa:reference interface="com.alipay.demo.SampleService" id="sampleService"/>
<sofa:reference interface="com.alipay.demo.SampleService" id="sampleService">
  <sofa:binding.bolt/>
</sofa:reference>
```

仅仅做到上下文隔离就足够吗？



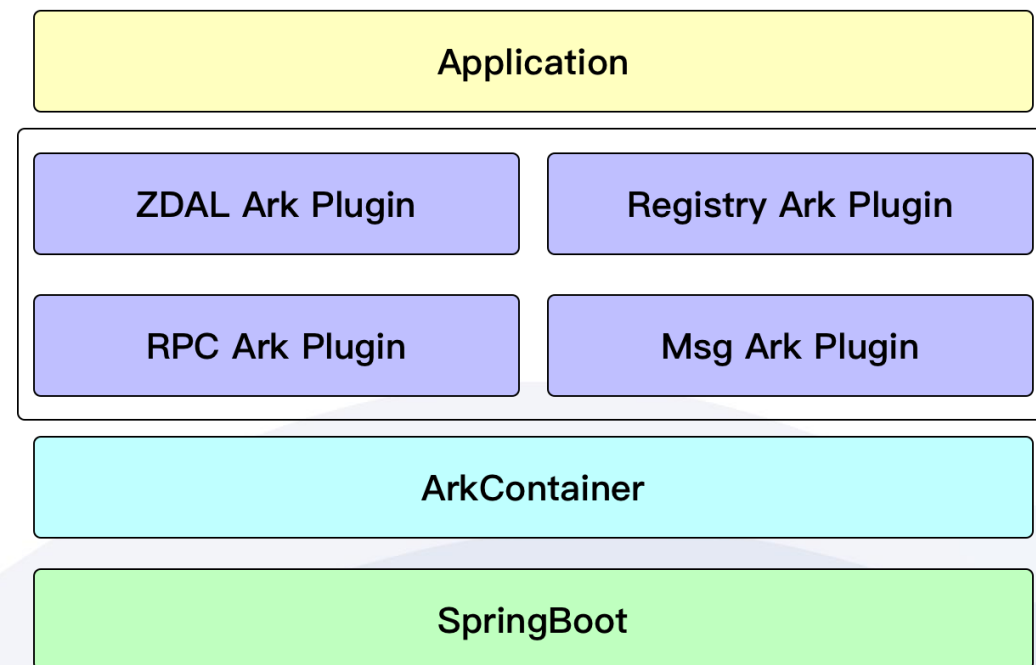
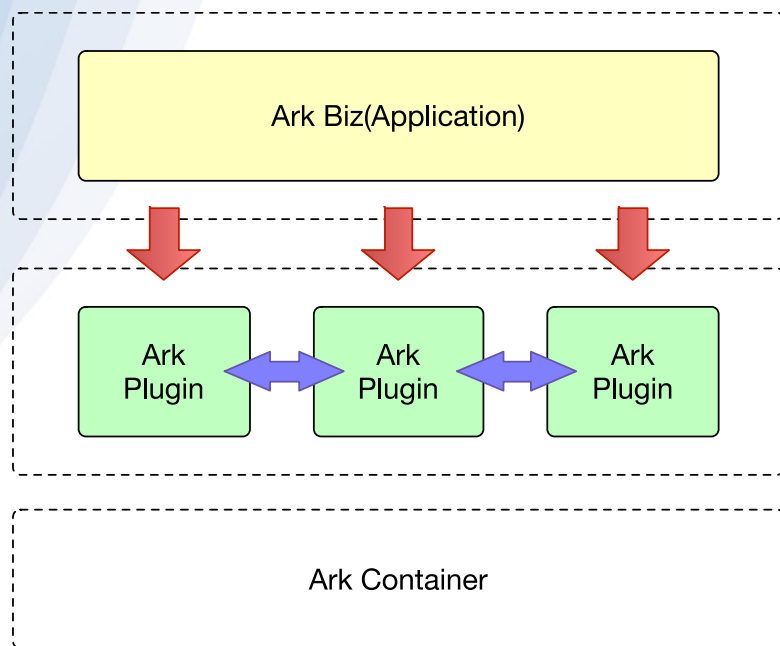
Dependency Hell

NoSuchMethodError

ClassNotFound

NoClassDefFoundError

SOFA 进一步类隔离



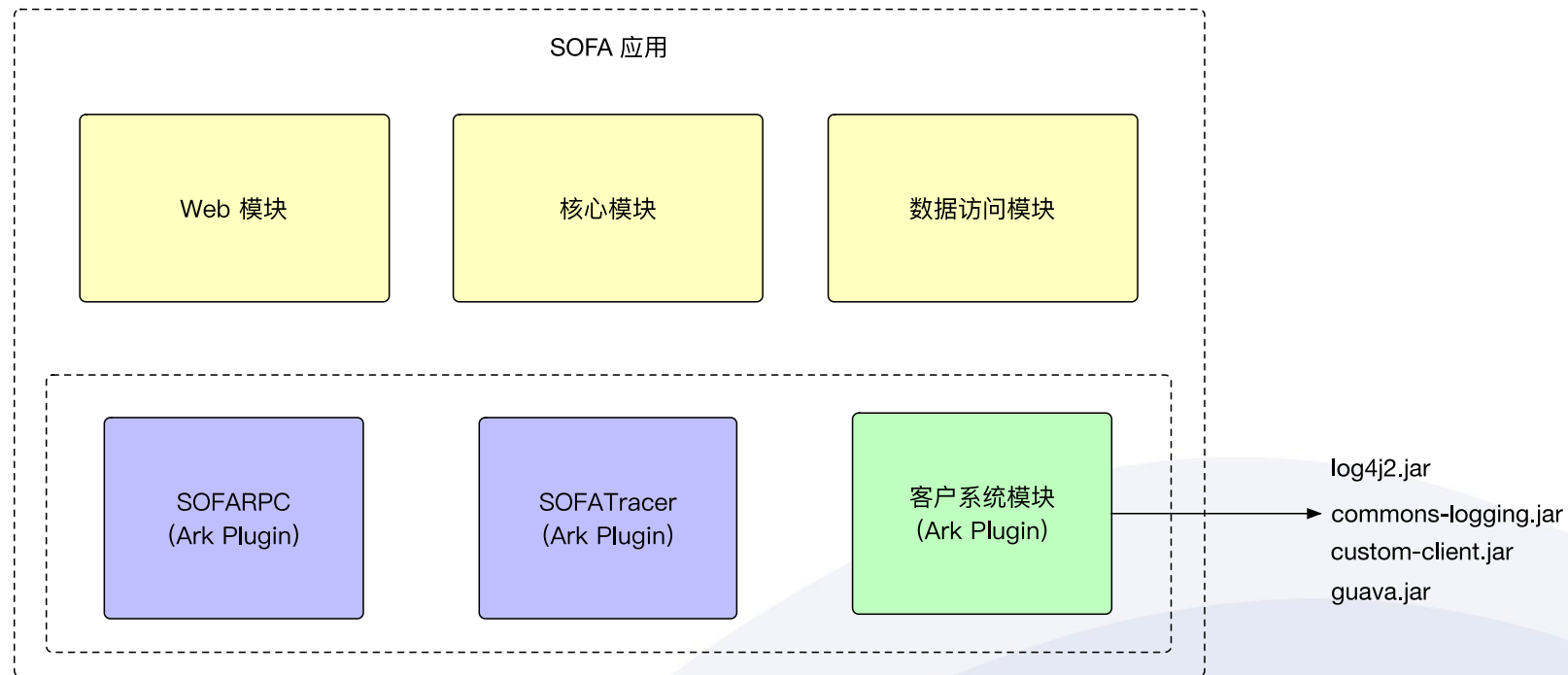
优势

减少中间件和业务代码的基础类库的冲突

让中间件之间减少类库冲突 保持业务编码的简单性

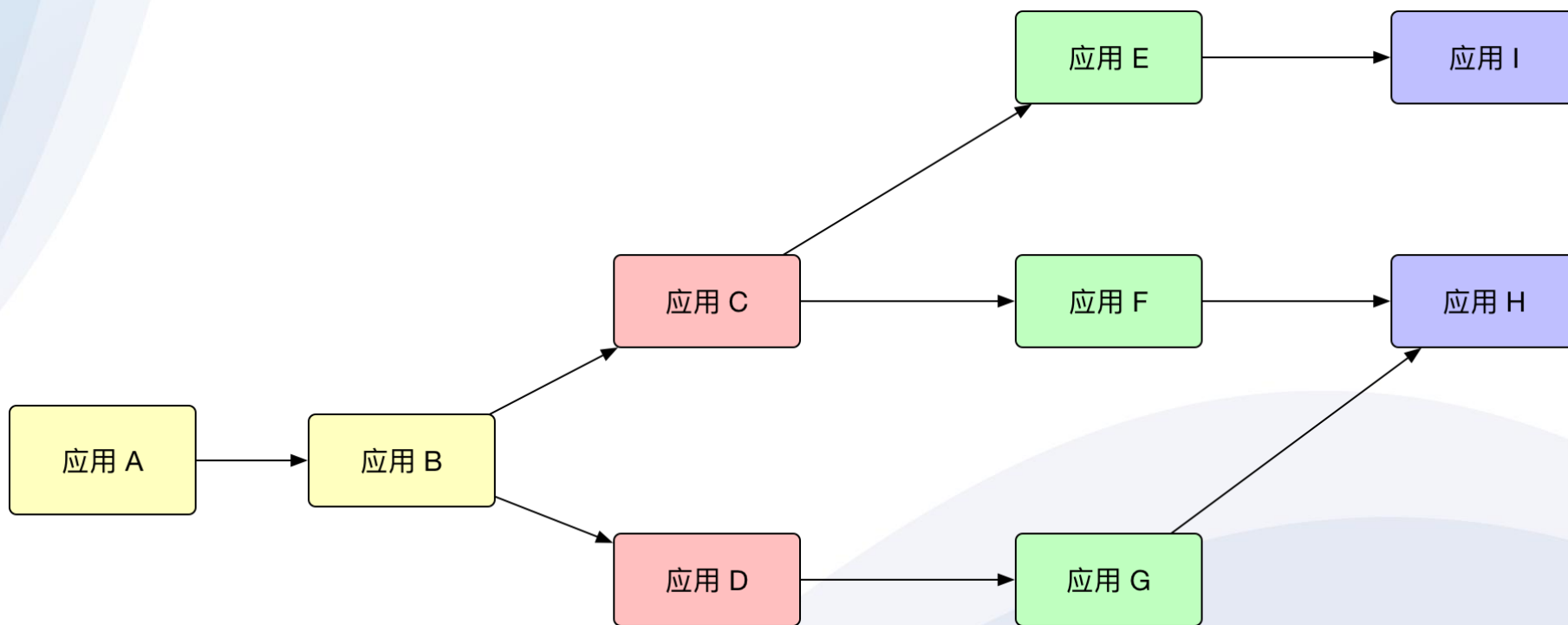


类隔离在业务系统中的使用



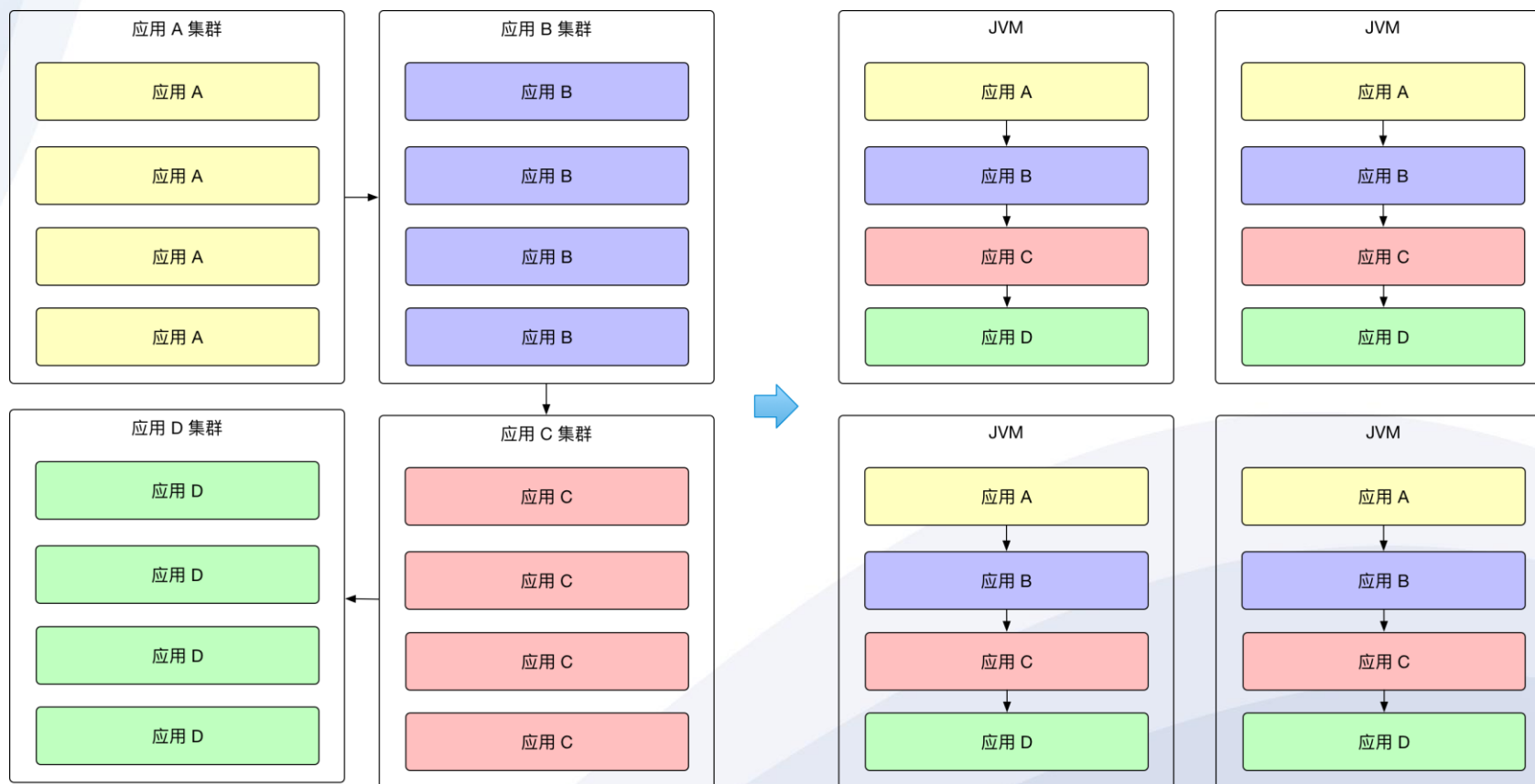
富客户端也适合使用类隔离来和宿主系统做隔离

大规模服务化之后的问题

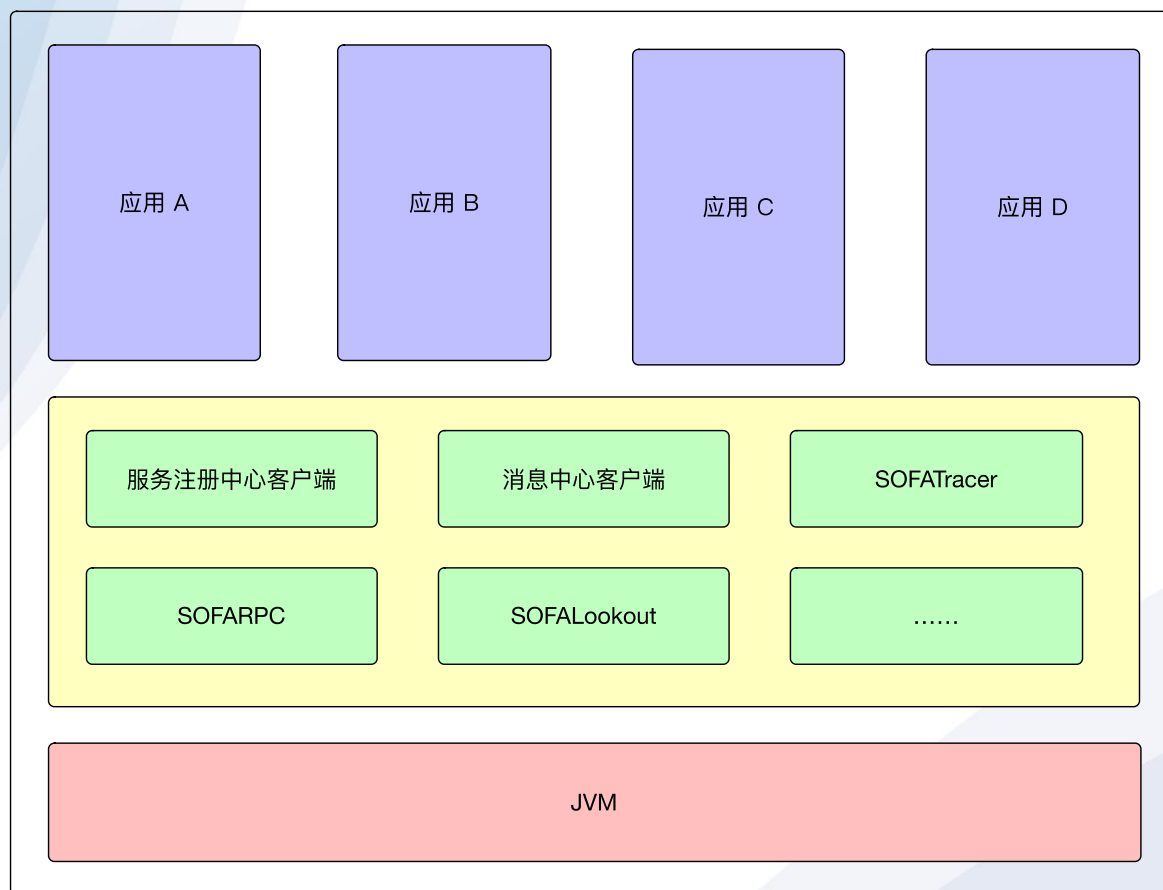


核心业务调用链路过深

SOFA --- 合并部署



SOFA 合并部署

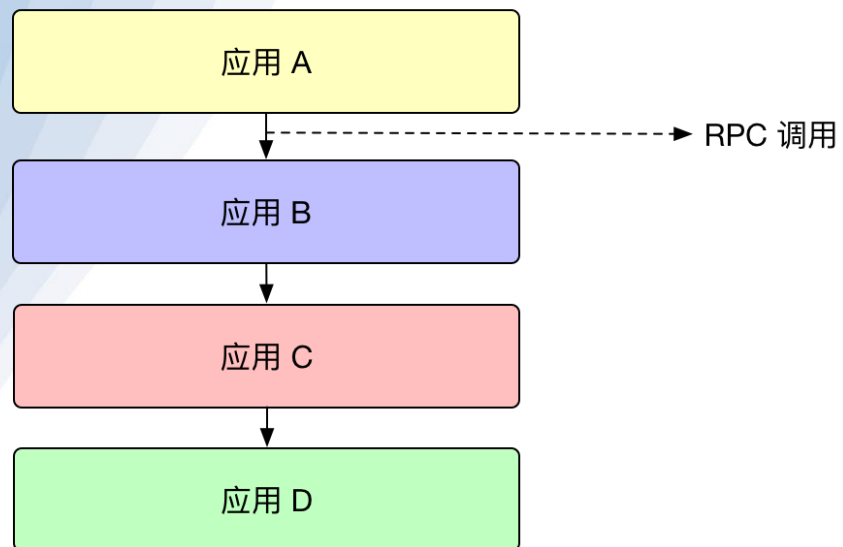


每个应用一个单独的 ClassLoader
防止出现类冲突

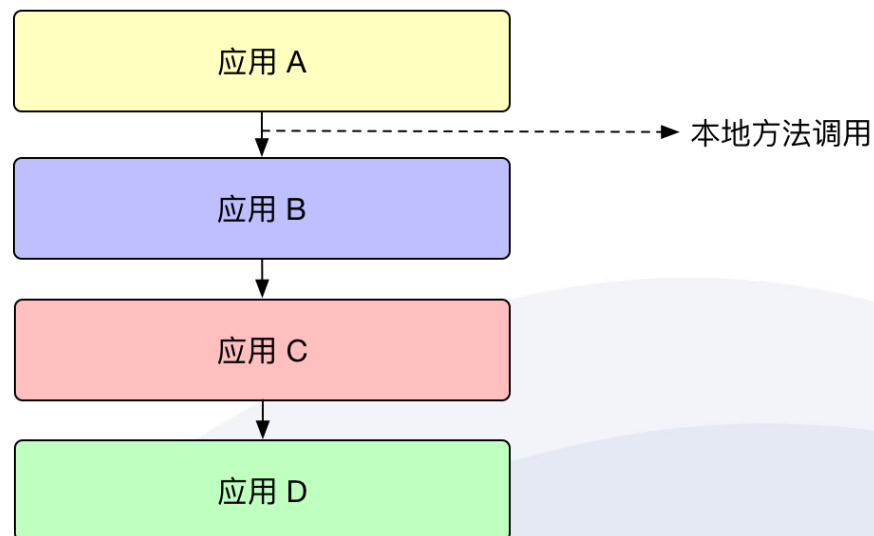
中间件的 ClassLoader
和应用的进行隔离

SOFA 合并部署

非合并部署



合并部署



用对象深度克隆
解决类隔离问题

SOFA 合并部署

```
public Object deepClone(Object instance, ClassLoader targetClassLoader) {
```

Source
ClassLoader

1. 以 targetClassLoader 创建类型实例
2. 深度遍历类型的字段
3. 简单字段直接从 instance 中取值，反射赋值
4. 复杂字段递归调用 deepClone 来赋值

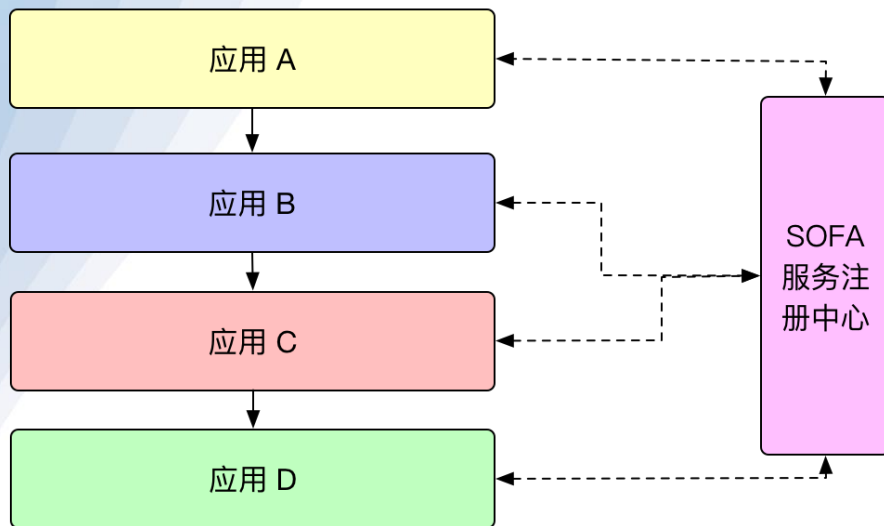
Target
ClassLoader

```
return clone;
```

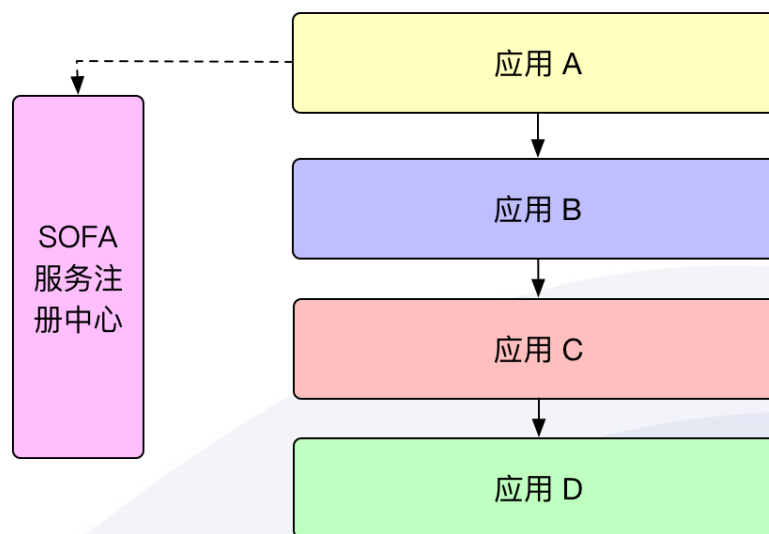
```
}
```

SOFA 合并部署

非合并部署



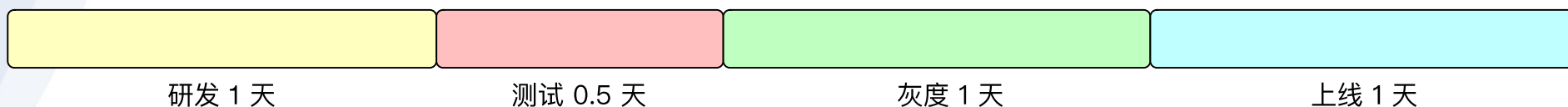
合并部署



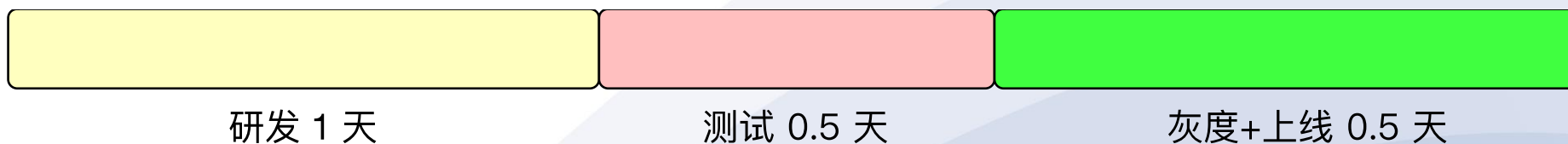
门面系统对外提供服务

非门面系统
不对外提供服务

仅仅做到类隔离就够了吗？



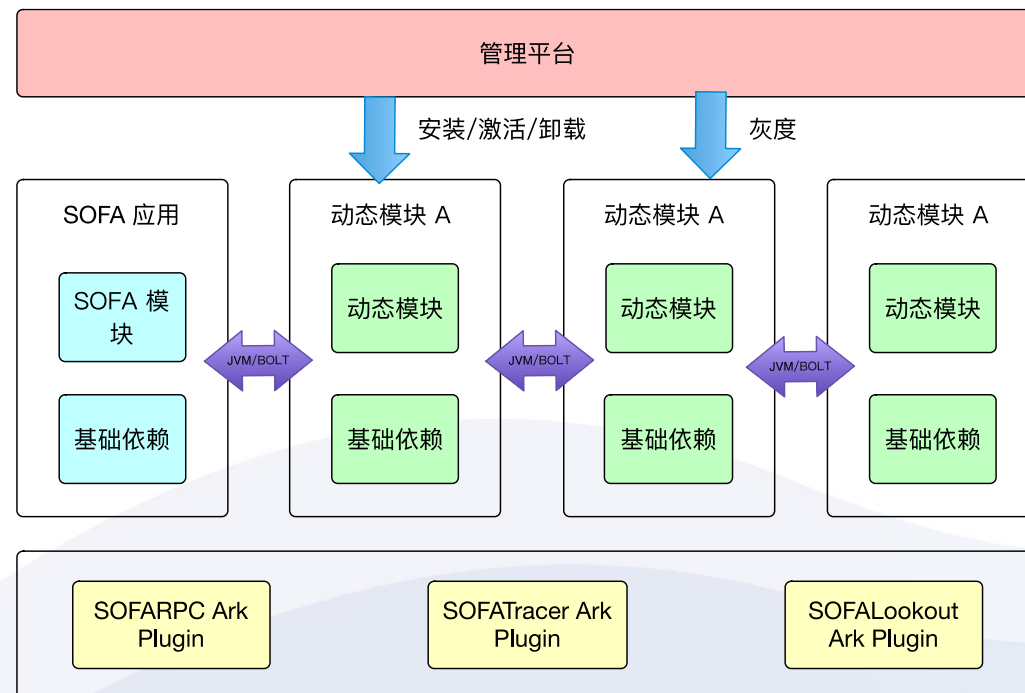
变更频繁的非主链路业务可以加快速度吗？



SOFA 模块动态化

动态模块：

- SOFA 应用可以动态地装载卸载 SOFA 模块。
- 提供和普通 SOFA 模块基本上一致的编程模型。
- 同一模块可以在应用中存在多个版本。



动态模块 --- 上车容易下车难

JDK 对于类的回收**非常严格**

- 该类所有实例已经被回收
- 加载该类的 ClassLoader 已经被回收
- 该类对应的 java.lang.Class 没有被任何地方引用

AppenderSkeleton 的 finalize 方法导致 log4j 无法正常被卸载：

```
public void finalize() {  
    if (!this.closed) {  
        LogLog.debug(msg: "Finalizing appender named [" + this.name + "].");  
        this.close();  
    }  
}
```

常见问题：

- 另外一个模块持有该模块的反射缓存。
- 自定义 finalize 方法。
- 自定义线程 while(true) 循环

解决方法：

- 明白常见的坑，不要主动跳坑里。
- 重复装载测试，观察 Metaspace 变化。

总结

普通模块化

- 仅仅做了模块的物理隔离。
- 业务不是很复杂的时候使用

SOFA 类隔离

- 在运行时隔离基础上，进一步提供类隔离。
- 需要克制地使用。
- 可以用来实现合并部署

SOFA 上下文隔离

- 进一步做上下文隔离，做到了部分的运行时隔离。
- 业务较为复杂的时候可以使用

SOFA 动态模块

- 满足非主链路变更频繁的业务的需求。
- 需要非常注意模块的卸载问题。

<https://gitee.com/alipay>
<https://github.com/alipay>



金融级分布式架构



ServiceMesh

欢迎关注微信公众号，获取更多技术干货！

