AOP 是什么

> 在软件业，AOP 为 Aspect Oriented Programming 的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP 是 OOP 的延续，是软件开发中的一个热点，是函数式编程的一种衍生范型。利用 AOP 可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

它是一种关注点分离的技术。我们软件开发时经常提一个词叫做“业务逻辑”或者“业务功能”，我们的代码主要就是实现某种特定的业务逻辑。但是我们往往不能专注于业务逻辑，比如我们写业务逻辑代码的同时，还要写事务管理、缓存、日志等等通用化的功能，而且每个业务功能都要和这些业务功能混在一起，非常非常地痛苦。为了将业务功能的关注点和通用化功能的关注点分离开来，就出现了 AOP 技术。

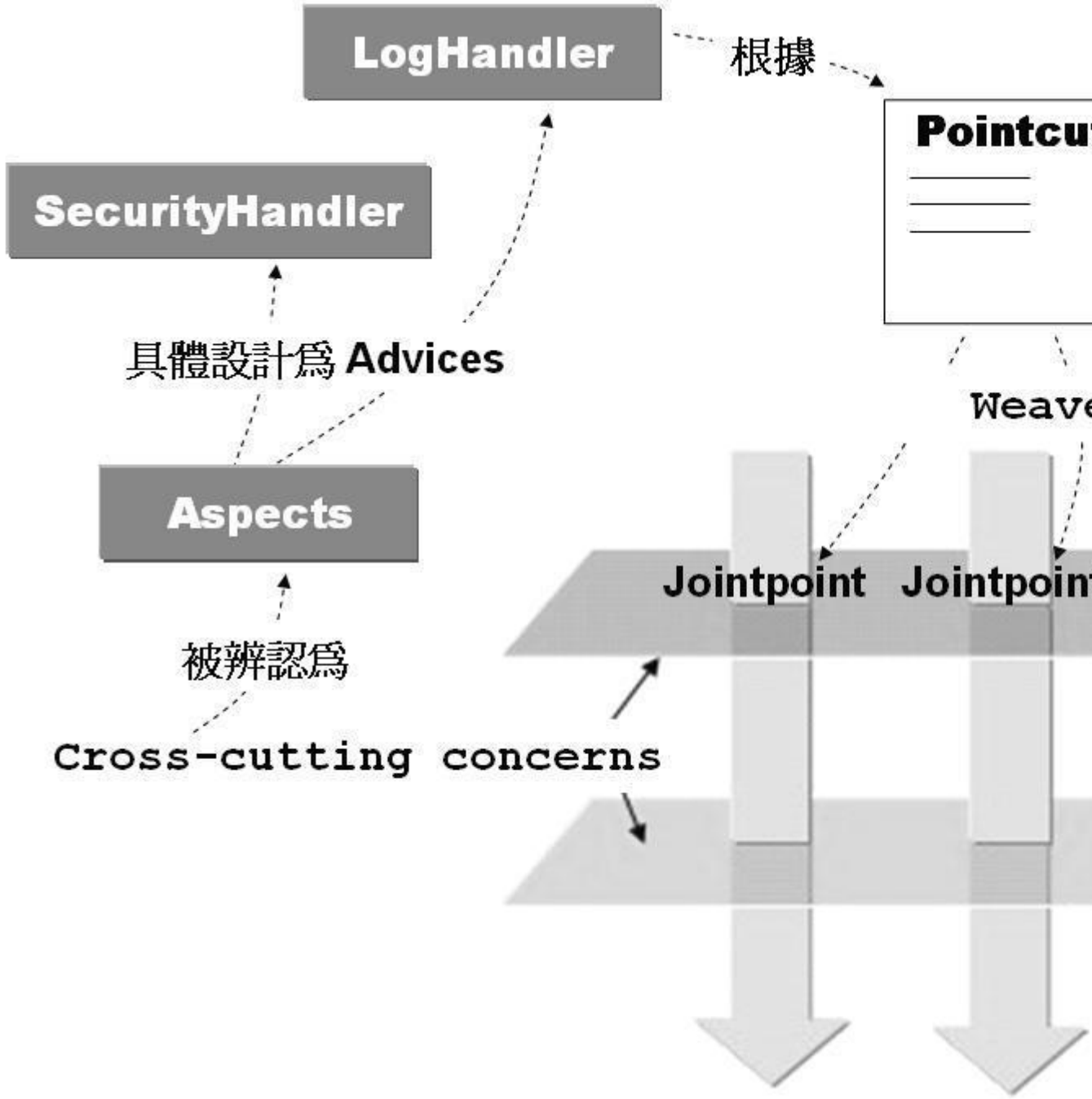
AOP 和 OOP

面向对象的特点是继承、多态和封装。为了符合单一职责的原则，OOP 将功能分散到不同的对象中去。让不同的类设计不同的方法，这样代码就分散到一个个的类中。可以降低代码的复杂程度，提高类的复用性。

但是在分散代码的同时，也增加了代码的重复性。比如说，我们在两个类中，可能都需要在每个方法中做日志。按照 OOP 的设计方法，我们就必须在两个类的方法中都加入日志的内容。也许他们是完全相同的，但是因为 OOP 的设计让类与类之间无法联系，而不能将这些重复的代码统一起来。然而 AOP 就是为了解决这类问题而产生的，它是在运行时动态地将代码切入到类的指定方法、指定位置上的编程思想。

如果说，面向过程的编程是一维的，那么面向对象的编程就是二维的。OOP 从横向上区分出一个个的类，相比过程式增加了一个维度。而面向切面结合面向对象

编程是三维的，相比单单的面向对象编程则又增加了“方面”的维度。从技术上来说，AOP 基本上是通过代理机制实现的。



AOPConcept.JPG

AOP 在 Android 开发中的常见用法

我封装的 library 已经把常用的 Android AOP 用法概况在其中

github 地址：https://github.com/fengzhizi715/SAF-AOP

0. 下载和安装

在根目录下的 build.gradle 中添加

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath
'com.hujiang.aspectjx:gradle-android-plugin-aspectjx:1.0.8'
    }
}
```

在 app 模块目录下的 build.gradle 中添加

```
apply plugin: 'com.hujiang.android-aspectjx'

...

dependencies {
    compile 'com.saffframework:saf-aop:1.0.0'
    ...
}
```

1. 异步执行 app 中的方法

告别 Thread、Handler、BroadCoast 等方式更简单的执行异步方法。只需在目标方法上标注@Async

```
import android.app.Activity;import android.os.Bundle;import
android.os.Looper;import android.widget.Toast;
import com.saframework.app.annotation.Async;import
com.saframework.log.L;
/**
 * Created by Tony Shen on 2017/2/7.
 */
public class DemoForAsyncActivity extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        initData();
    }

    @Async
    private void initData() {

        StringBuilder sb = new StringBuilder();
        sb.append("current
thread=").append(Thread.currentThread().getId())
                .append("\r\n")
                .append("ui thread=")
                .append(Looper.getMainLooper().getThread().getId());


        Toast.makeText(DemoForAsyncActivity.this, sb.toString(),
Toast.LENGTH_SHORT).show();
        L.i(sb.toString());
    }
}
```

可以清晰地看到当前的线程和 UI 线程是不一样的。

@Async 执行结果.png

@Async 的原理如下，借助 Rxjava 实现异步方法。

```
import android.os.Looper;
import org.aspectj.lang.ProceedingJoinPoint;import
org.aspectj.lang.annotation.Around;import
org.aspectj.lang.annotation.Aspect;import
org.aspectj.lang.annotation.Pointcut;
import rx.Observable;import rx.Subscriber;import
rx.android.schedulers.AndroidSchedulers;import
rx.schedulers.Schedulers;
/**
 * Created by Tony Shen on 16/3/23.
 */@Aspectpublic class AsyncAspect {

    @Around("execution(!synthetic * *(..)) && onAsyncMethod()")
    public void doAsyncMethod(final ProceedingJoinPoint joinPoint)
throws Throwable {
        asyncMethod(joinPoint);
    }


@Pointcut("@within(com.saffframework.app.annotation.Async)||@annotatio
n(com.saffframework.app.annotation.Async)")
    public void onAsyncMethod() {
    }

    private void asyncMethod(final ProceedingJoinPoint joinPoint) throws
Throwable {
```

```
        Observable.create(new Observable.OnSubscribe<Object>() {

            @Override
            public void call(Subscriber<? super Object> subscriber) {
                Looper.prepare();
                try {
                    joinPoint.proceed();
                } catch (Throwable throwable) {
                    throwable.printStackTrace();
                }
                Looper.loop();
            }
        }).subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.ma
inThread()).subscribe();
    }
}
```

2. 将方法返回的结果放于缓存中

我先给公司的后端项目写了一个 CouchBase 的注解，该注解是借助 Spring Cache 和 CouchBase 结合的自定义注解，可以把某个方法返回的结果直接放入 CouchBase 中，简化了 CouchBase 的操作。让开发人员更专注于业务代码。

受此启发，我写了一个 Android 版本的注解，来看看该注解是如何使用的。

```
import android.app.Activity;import android.os.Bundle;import
android.widget.Toast;
import com.saframework.app.annotation.Cacheable;import
com.saframework.app.domain.Address;import
com.saframework.cache.Cache;import
com.saframework.injectview.Injector;import
com.saframework.injectview.annotations.OnClick;import
com.saframework.log.L;import
com.saframwork.tony.common.utils.StringUtils;
/**
 * Created by Tony Shen on 2017/2/7.
 */
public class DemoForCacheableActivity extends Activity {
```

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_demo_for_cacheable);
        Injector.injectInto(this);

        initData();
    }

    @Cacheable(key = "address")
    private Address initData() {

        Address address = new Address();
        address.country = "China";
        address.province = "Jiangsu";
        address.city = "Suzhou";
        address.street = "Ren min Road";

        return address;
    }

    @OnClick(id={R.id.text})
    void clickText() {

        Cache cache = Cache.get(this);
        Address address = (Address) cache.getObject("address");
        Toast.makeText(this,
StringUtils.printObject(address),Toast.LENGTH_SHORT).show();
        L.json(address);
    }
}
```

在 initData() 上标注 @Cacheable 注解和缓存的 key，点击 text 按钮之后，就会打印出缓存的数据和 initData() 存入的数据是一样的。

@Cacheable 执行结果.png

目前，该注解 @Cacheable 只适用于 Android 4.0 以上。

3. 将方法返回的结果放入 SharedPreferences 中

该注解 @Prefs 的用法跟上面 @Cacheable 类似，区别是将结果放到 SharedPreferences。

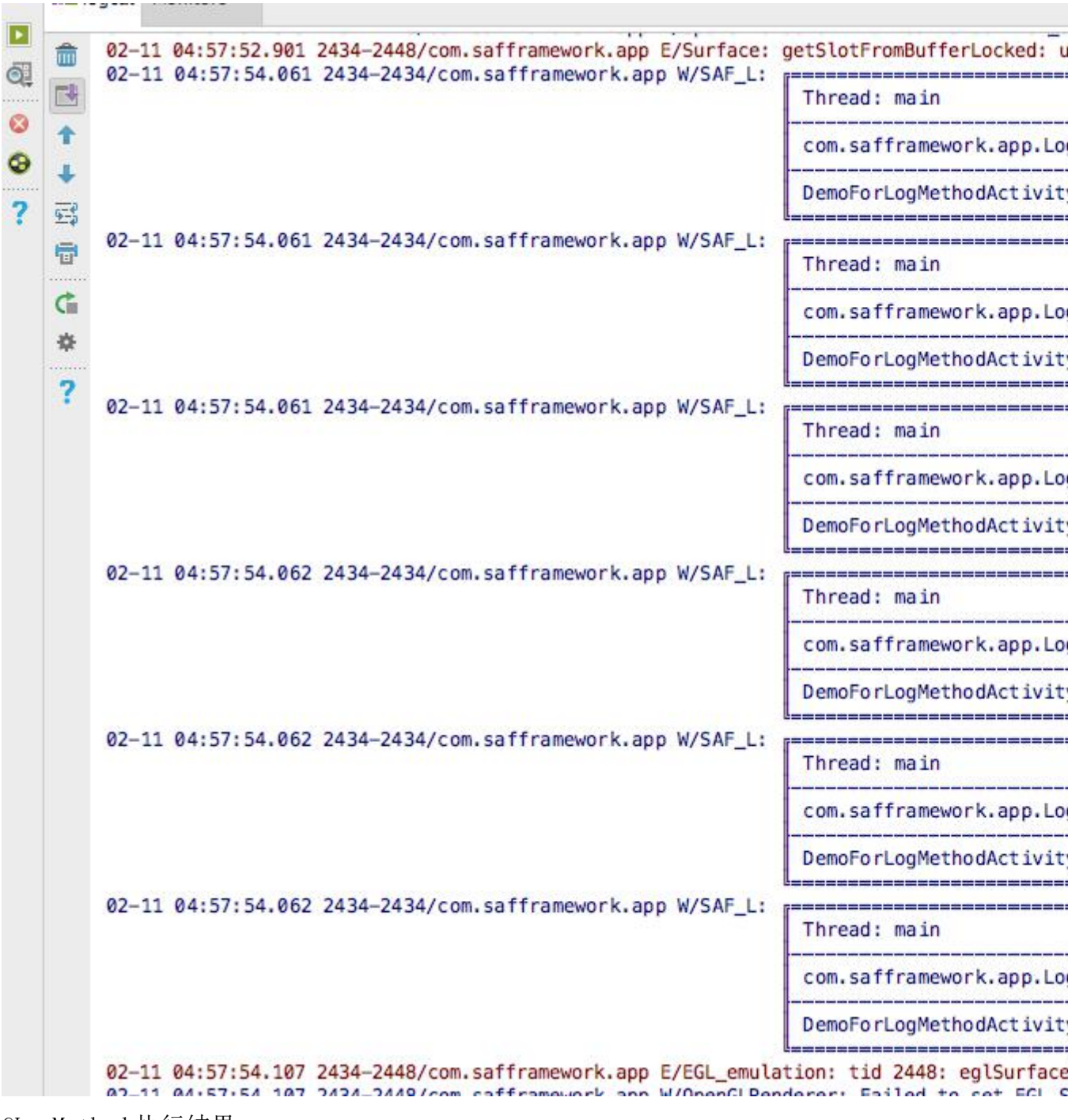
同样，该注解 @Prefs 也只适用于 Android 4.0 以上

4. App 调试时，将方法的入参和出参都打印出来

在调试时，如果一眼无法看出错误在哪里，那肯定会把一些关键信息打印出来。
在 App 的任何方法上标注 @LogMethod，可以实现刚才的目的。

```
public class DemoForLogMethodActivity extends Activity{
```

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        initData1();

        initData2("test");

        User u = new User();
        u.name = "tony";
        u.password = "123456";
        initData3(u);
    }

    @LogMethod
    private void initData1() {
    }

    @LogMethod
    private String initData2(String s) {

        return s;
    }

    @LogMethod
    private User initData3(User u) {

        u.password = "abcdefg";

        return u;
    }
}
```



@LogMethod 执行结果.png

目前，方法的入参和出参只支持基本类型和 String，未来我会加上支持任意对象的打印以及优雅地展现出来。

5. 在调用某个方法之前、以及之后进行 hook

通常，在 App 的开发过程中会在一些关键的点击事件、按钮、页面上进行埋点，方便数据分析师、产品经理在后台能够查看和分析。

以前在大的电商公司，每次 App 发版之前，都要跟数据分析师一起过一下看看哪些地方需要进行埋点。发版在即，添加代码会非常仓促，还需要安排人手进行测试。而且埋点的代码都很通用，所以产生了 @Hook 这个注解。它可以在调用某个方法之前、以及之后进行 hook。可以单独使用也可以跟任何自定义注解配合使用。

```
@HookMethod(beforeMethod = "method1",afterMethod = "method2")
private void initData() {

    L.i("initData()");
}

private void method1() {
    L.i("method1() is called before initData()");
}

private void method2() {
    L.i("method2() is called after initData()");
}
```

来看看打印的结果，不出意外先打印 method1() is called before initData()，再打印 initData()，最后打印 method2() is called after initData()。

@Hook 执行的结果.png

@Hook 的原理如下, beforeMethod 和 afterMethod 即使找不到或者没有定义也不会影响原先方法的使用。

```
import com.saffframework.app.annotation.HookMethod;import
com.saffframework.log.L;import
com.saffframwork.tony.common.reflect.Reflect;import
com.saffframwork.tony.common.reflect.ReflectException;import
com.saffframwork.tony.common.utils.Preconditions;
import org.aspectj.lang.ProceedingJoinPoint;import
org.aspectj.lang.annotation.Around;import
org.aspectj.lang.annotation.Aspect;import
org.aspectj.lang.annotation.Pointcut;import
org.aspectj.lang.reflect.MethodSignature;
import java.lang.reflect.Method;

/**
 * Created by Tony Shen on 2016/12/7.
 */@Aspectpublic class HookMethodAspect {

    @Around("execution(!synthetic * *(..)) && onHookMethod()")
```

```
    public void doHookMethodd(final ProceedingJoinPoint joinPoint)
throws Throwable {
        hookMethod(joinPoint);
    }


@Pointcut("@within(com.saframework.app.annotation.HookMethod)||@anno
tation(com.saframework.app.annotation.HookMethod)")
    public void onHookMethod() {
    }

    private void hookMethod(final ProceedingJoinPoint joinPoint) throws
Throwable {
        MethodSignature signature = (MethodSignature)
joinPoint.getSignature();
        Method method = signature.getMethod();

        HookMethod hookMethod = method.getAnnotation(HookMethod.class);

        if (hookMethod==null) return;

        String beforeMethod = hookMethod.beforeMethod();
        String afterMethod = hookMethod.afterMethod();

        if (Preconditions.isNotBlank(beforeMethod)) {
            try {
                Reflect.on(joinPoint.getTarget()).call(beforeMethod);
            } catch (ReflectException e) {
                e.printStackTrace();
                L.e("no method "+beforeMethod);
            }
        }

        joinPoint.proceed();

        if (Preconditions.isNotBlank(afterMethod)) {
            try {
                Reflect.on(joinPoint.getTarget()).call(afterMethod);
            } catch (ReflectException e) {
                e.printStackTrace();
                L.e("no method "+afterMethod);
            }
        }
    }
```

```
}
```

6. 安全地执行方法，不用考虑异常情况

一般情况，写下这样的代码肯定会抛出空指针异常，从而导致 App Crash。

```
private void initData() {

    String s = null;
    int length = s.length();
}
```

然而，使用 @Safe 可以确保即使遇到异常，也不会导致 App Crash，给 App 带来更好的用户体验。

```
@Safe
private void initData() {

    String s = null;
    int length = s.length();
}
```

再看一下 logcat 的日志，App 并没有 Crash 只是把错误的日志信息打印出来。

logcat 的日志.png

我们来看看，@Safe 的原理，在遇到异常情况时直接 catch Throwable。

```
import com.saffframework.log.L;
import org.aspectj.lang.ProceedingJoinPoint;import
org.aspectj.lang.annotation.Around;import
org.aspectj.lang.annotation.Aspect;import
org.aspectj.lang.annotation.Pointcut;
import java.io.PrintWriter;import java.io.StringWriter;
/**
 * Created by Tony Shen on 16/3/23.
 */@Aspectpublic class SafeAspect {

    @Around("execution(!synthetic * *(..)) && onSafe()")
    public Object doSafeMethod(final ProceedingJoinPoint joinPoint)
throws Throwable {
        return safeMethod(joinPoint);
    }
```

```
@Pointcut("@within(com.saffframework.app.annotation.Safe)||@annotation
(com.saffframework.app.annotation.Safe)")
```

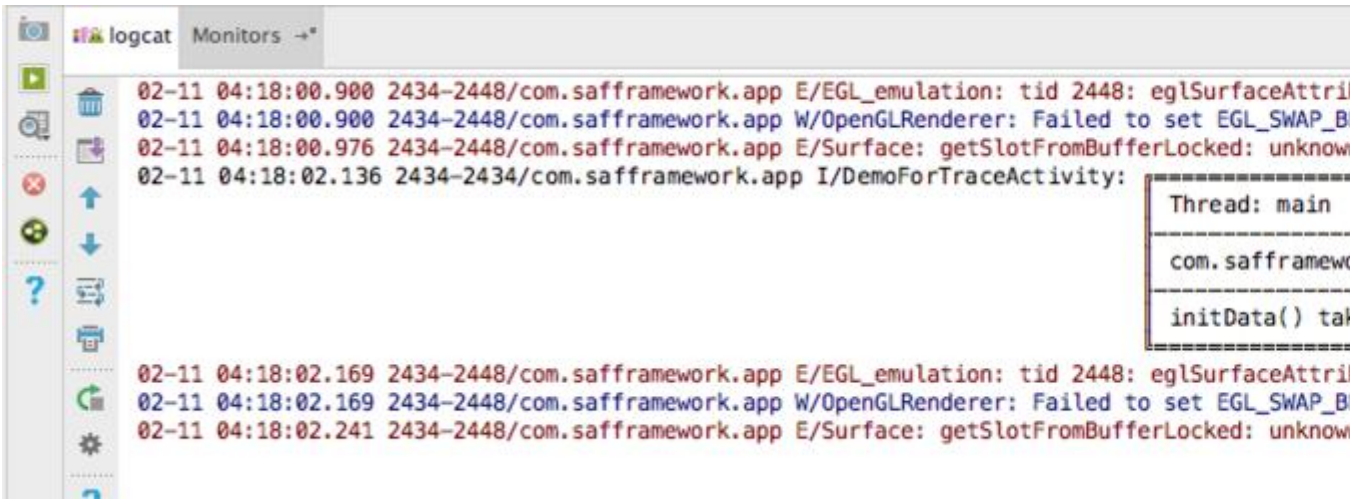
```
    public void onSafe() {
    }

    private Object safeMethod(final ProceedingJoinPoint joinPoint)
throws Throwable {

        Object result = null;
        try {
            result = joinPoint.proceed(joinPoint.getArgs());
        } catch (Throwable e) {
            L.w(getStringFromException(e));
        }
        return result;
    }

    private static String getStringFromException(Throwable ex) {
        StringWriter errors = new StringWriter();
        ex.printStackTrace(new PrintWriter(errors));
        return errors.toString();
    }
}
```

7. 追踪某个方法花费的时间，用于性能调优

无论是开发 App 还是 Service 端，我们经常会用做一些性能方面的测试，比如查看某些方法的耗时。从而方便开发者能够做一些优化的工作。@Trace 就是为这个目的而产生的。

```
    @Trace
    private void initData() {

        for (int i=0;i<10000;i++) {
            Map map = new HashMap();
            map.put("name","tony");
            map.put("age","18");
            map.put("gender","male");
        }
    }
```

来看看，这段代码的执行结果，日志记录花费了 3ms。



@Trace 执行结果.png

只需一个@Trace 注解，就可以实现追踪某个方法的耗时。如果耗时过长那就需要优化代码，优化完了再进行测试。
当然啦，在生产环境中不建议使用这样的注解。

总结

AOP 是 OOP 的有力补充。玩好 AOP 对开发 App 是有很大的帮助的，当然也可以直接使用我的库：），而且新的使用方法我也会不断地更新。由于水平有限，如果有任何地方阐述地不正确，欢迎指出，我好及时修改：）