前面我们已经介绍过 APM 链路监控实现的原理，这篇文章会介绍具体的代码实现。

0x01 使用更强大 AdviceAdapter 来生成 try-catch-finally 语句块

函数注入其实就是对其内容用 try-catch-finally 包裹起来。
考虑一个实际的场景，我们经常会在代码内部捕获很多异常，甚至是全局捕获异常，这样的情况下很多异常被淹没了，因此有必要对代码内部捕获的异常也做记录和上报，在上报端进行判断是否需要处理。

字节码要注入的效果如下

```
源代码
public void saveFile() {
    try {
        fileSaveService.save();
    } catch (FileSaveException e) {
        e.printStackTrace();
    }
    doOtherBusiness();
}

字节码改写以后

public void saveFile() {
    List<Throwable> caughtThrowableList = new
ArrayList<>();
    try {

Tracer.getInstance().startSubSpan("saveFile()",
"geek01-demo", Span.SpanType.LOCAL_ONLY);
```

```
        try {
            fileSaveService.save();
        } catch (FileSaveException e) {
            caughtThrowableList.add(e);
            e.printStackTrace();
        }
        doOtherBusiness();
    } catch (Throwable e) {
        Tracer.getInstance().completeSubSpan(e,
caughtThrowableList);
        throw e;
    }
    Tracer.getInstance().completeSubSpan(null,
caughtThrowableList);
}
```

这里采用继承 ASM-commons 包中的 AdviceAdapter 类。这个方法适配器是一个抽象类，可以用于在方法的开始和 RETURN、ATHROW 指令前插入代码，它的 onMethodEnter、onMethodExit、visitMaxs 函数是比较简单的切入点。

```
public class TraceClassTransform extends
AdviceAdapter {
    @Override
    protected void onMethodEnter() {

    }
    @Override
    protected void onMethodExit(int opcode) {

    }
    @Override
    public void visitMaxs(int maxStack, int
maxLocals) {

    }
}
```

onMethodEnter

这个方法在函数调用进入时执行，可以做初始化的操作。

- 在函数开始时进行变量的初始化，首先新建 caughtThrowableList 变量。对应的字节码是变量初始化三部曲 new-dup-invokespecial<init>
- 增加一个 try 语句开始的 label，为后面完整的构造 try-catch 块做准备，具体的指令是visitLabel(methodStartLabel)
- 接下来调用Tracer.getInstance().startSubSpan()函数在 ThreadLocal 的栈中入栈当前 span 调用，这部分完整的代码可以看文末 github 仓库完整代码

```
@Override
protected void onMethodEnter() {
    visitTypeInsn(NEW, "java/util/ArrayList");
    dup();
    visitMethodInsn(INVOKESPECIAL,
"java/util/ArrayList", "<init>", "()V", false);
    storeLocal(caughtExceptionListLocal,
Type.getObjectType("java/util/ArrayList"));

    visitLabel(methodStartLabel);
    push(appId);
    push(className);
    push(methodName);
    invokeStatic(Type.getType(this.getClass()),
            new Method("startSubSpan",
                    "
(Ljava/lang/String;Ljava/lang/String;Ljava/lang/S
tring;)V"
            ));
}
```

onMethodExit

我们在这里处理正常退出的情况，异常退出的情况，我们统一在 visitMaxs 中处理。

```
 @Override
protected void onMethodExit(int opcode) {
    if (opcode != ATHROW) {
        exitMethod(false);
    }
}

private void exitMethod(boolean throwing) {
    if (throwing) {
        dup();
    } else {
        push((Type) null);
    }
    loadLocal(caughtExceptionListLocal);

    invokeStatic(Type.getType(this.getClass()),
            new Method("completeSubSpan",
                    "
(Ljava/lang/Object;Ljava/lang/Object;)V"
            ));
    if (throwing) {
        visitInsn(ATHROW);
    }
}
```

正常退出的情况下，需要执行 Tracer.getInstance().completeSubSpan(null, caughtThrowableList);只用在字节码中把 completeSubSpan 所需的参数入栈，调用 invokeStatic 指令就可以了。第一个参数未捕获异常为 null，第二个参数捕获异常列表为 caughtExceptionListLocal

visitMaxs

在这里真正处理异常退出的情
况，Tracer.getInstance().completeSubSpan(uncaughtExcaughtThrowableList); 。使用 dup 指令复制栈顶现在的元素（未捕获异常），如果不这样的话，使用 invokeStatic 指令调用 completeSubSpan 以后，未捕获异常就再也找不到了，没有办法调用 athrow 把异常抛出去

```
@Override
public void visitMaxs(int maxStack, int
maxLocals) {
    exitMethod(true);
    super.visitMaxs(maxStack, maxLocals);
}
```

visitTryCatchBlock 与 visitLabel

到目前为止还没有处理代码块内部捕获异常的逻辑。这里只需要简单的把内部捕获的异常添加到 caughtExceptionList 中即可，为了能进入 catch 代码块内部，需要覆写 visitLabel 方法。同时因为本身代码块会包裹 try-catch-finally 语句块，需要过滤掉刚才自己添加的这部分指令。采用的方式是用一个 HashMap 代码块内部的 catch 块

```
@Override
public void visitLabel(Label label) {
    super.visitLabel(label);
    if (label != null && matchedHandle.get(label)
!= null && !label.equals(endFinallyLabel)) {

        dup(); // exception

        loadLocal(caughtExceptionListLocal,
Type.getObjectType("java/util/ArrayList"));
        swap(); // swap exception <-> list

invokeVirtual(Type.getType(ArrayList.class), new
Method("add", "(Ljava/lang/Object;)Z"));
        pop();
    }
}

@Override
public void visitTryCatchBlock(Label start, Label
end, Label handler, String exception) {
    super.visitTryCatchBlock(start, end, handler,
exception);
    if (exception != null) {
        List<String> handles =
matchedHandle.get(handler);
        if (handles == null) handles = new
ArrayList<>();
        handles.add(exception);
        matchedHandle.put(handler, handles);
    }
}
```

之前写这段代码的时候调试了很久没有成功，后来定位到是invokeVirtual(Type.getType(ArrayList.class), new Method("add", "(Ljava/lang/Object;)Z")); 语句后面少了一个 pop 指令，为什么需要 pop 指令呢？ArrayList add 函数的函数声明为 public boolean add(E e) { }，它有一个 boolean 类型的返回值，但是正常的使用中几乎从来没有人知道或者使用过这个返回值，导致调用完栈帧没有清空，造成了后续一连串指令的错乱。

0x02 获取特定的上下文信息

在注入 Tomcat、JDBC、Jedis等函数库时，经常需要动态的取一些重要的上线文信息，比如请求的 url、HTTP 方法、JDBC 执行的 SQL、Redis 执行的指令等。
以 Jedis 的 set 函数注入为例

```
public String set(final String key, String value)
{
    checkIsInMultiOrPipeline();
    client.set(key, value);
    return client.getStatusCodeReply();
}
```

set 函数的函数参数就是属于当前 span 非常重要的信息，否则只是上报一个简单的 Jedis.set() 调用，如果这个调用耗时很久，在排查问题时还是不知道具体是执行什么 Redis 的命令耗时较长。因此需要对具体的类做具体的处理，常见需要处理的库有： Jedis、各种连接池、Tomcat servlet、Dubbo provider、Mongo、JDBC

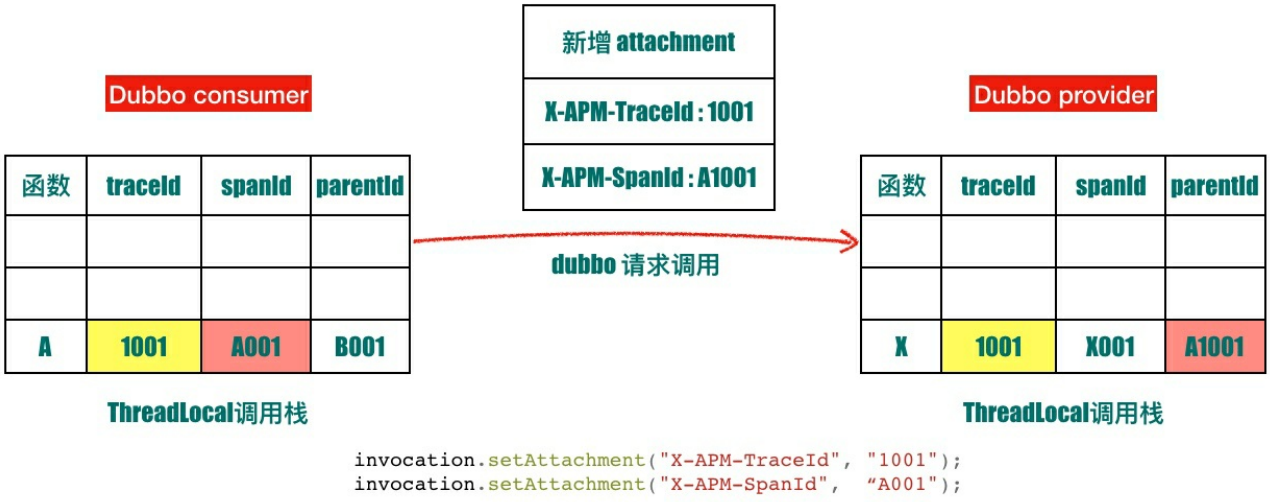
0x03 跨进程链路调用的实现

前面已经介绍过跨进程调用是采用在调用方注入当前 traceId 和 spanId 来实现的。以调用 Dubbo 为例，Dubbo 真正的远程调用是在 com.alibaba.dubbo.rpc.cluster.support.AbstractClus 函数

```
public Result invoke(Invocation invocation)
throws RpcException {
    return new RpcResult(doInvoke(proxy,
invocation.getMethodName(),
invocation.getParameterTypes(),
invocation.getArguments()));
}
```

需要把上述代码改写为

```
public Result invoke(Invocation invocation)
throws RpcException {
    invocation.setAttachment("X-APM-TraceId",
"traceId-1001");
    invocation.setAttachment("X-APM-SpanId",
"spanId-A001");
    return new RpcResult(doInvoke(proxy,
invocation.getMethodName(),
invocation.getParameterTypes(),
invocation.getArguments()));
}
```



实现示例代码如下

```
public static void addHeader(Span span, Object o)
{
    try {
        if (span.getTraceId() != null) {
            MethodUtils.invokeExactMethod(o,
"setAttachment", TraceHeaders.TRACE_ID,
span.getTraceId());
        }
        if (span.getSpanId() != null) {
            MethodUtils.invokeExactMethod(o,
"setAttachment", TraceHeaders.SPAN_ID,
span.getSpanId());
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

0x04 代码仓库

完整的 demo 代码我放在了 github 上：
https://github.com/arthur-zhang/geek01/tree/master/javaagent-demo

里面包含了实现一个 APM 系统最核心的部分：字节码改写、ThreadLocal 实现调用栈、跨进程调用。辅助的数据上报、告警、统计因为和业务比较强相关，这里没有放上来，如果感兴趣可以单独沟通。

0x05 番外篇：「Nginx、Node.js、安卓」我也要 APM

- 拥抱OpenResty
  OpenResty 是一个基于 Nginx 与 Lua 的高性能 Web 平台，其内部集成了大量精良的 Lua 库、第三方模块以及大多数的依赖项。用于方便地搭建能够处理超高并发、扩展性极高的动态 Web 应用、Web 服务和动态网关。
  使用 OpenResty 可以比较灵活的实现添加 header，获取耗时、状态码等信息，用少量的代码就可以把 Nginx 加入到 APM 链路中来

```
- 定义Headers
local X_APM_TRACE_ID = "X-APM-TraceId"
local X_APM_SPAN_ID = 'X-APM-SpanId'
local X_APM_SPAN_NAME = 'Nginx'

-- 生成Nginx Span Id
local ngx_span_id = string.gsub(uuid(), '-', '')

-- 从Header中，获取父Span信息
local ngx_span_parent = nil
if req_headers ~= nil then
    ngx_span_parent = req_headers[X_APM_SPAN_ID]
end

-- 向Header中，写入Nginx Span相关信息
local trace_id = req_headers[X_APM_TRACE_ID]
if trace_id == nil then
    trace_id = string.gsub(uuid(), '-', '')
    ngx.req.set_header(X_APM_TRACE_ID, trace_id)
end

ngx.req.set_header(X_APM_SPAN_ID, ngx_span_id)
ngx.req.set_header(X_APM_SPAN_NAME,
X_APM_SPAN_NAME)
ngx.req.set_header(X_APM_SAMPLED, X_APM_SAMPLED)
```

- Node.js 上成熟 APM pandora.js：释放潘多拉的魔盒

pandora.js 是 一个可管理、可度量、可追踪的 Node.js 应用管理器，通过最新 Node.js 的链路追踪技术，Pandora.js 对业界常用的模块进行埋点追踪，能够产出常见的 open-tracing 格式的打点逻辑，在一定的数据分析之后，可以让应用清晰的发现自己的瓶颈，从而进行数据优化和逻辑分析。非常强大，我们现在也在生产环境使用。

- 安卓 APM：巨头不再藏着掖着
  18 年 360 和微信接连开源他们在安卓上的 APM 解决方案，分别是 ArgusAPM (https://github.com/Qihoo360/ArgusAPM) 和 matrix (https://github.com/Tencent/matrix)。当前的监控范围包括：应用安装包大小，帧率变化，启动耗时，卡顿，慢方法，SQLite 操作优化，文件读写，内存泄漏等等。两三年前我们之前也自研过安卓端的 APM 系统，但是因为安卓的碎片化、兼容性问题等，一直没敢把自研的方案推到线上，随着两大巨头的开源，可以供我们更多参考学习，把移动端这重要一环也加入到 APM 链路中来

0x06 小结与思考题

这篇文章我们讲解了 APM 系统的实际代码实现，重要的知识点如下：
第一，使用 ASM 库的AdviceAdapter 可以更方便的生成 try-catch-finally 语句块，除了处理未捕获的异常，这篇文章也介绍了如何把代码内部捕获的异常已进行上报。
第二，以 Dubbo 为例介绍了跨进程 tcp 调用的如何传递 traceId、spanId，以这个为基础，可以扩展到 HTTP 调用、SofaRPC 调用等各式各样的远程调用方式。
第三，Nginx、Node.js、安卓的 APM 方案如何选型

0x07 思考

留一道思考题：安卓的 APM 系统的原理是什么？它的字节码改写发生在哪一个阶段?

欢迎你在留言区留言，和我一起讨论。