Arthas原理与使用

1. Arthas简介

1.1 背景与定位

Arthas 是阿里巴巴开源的 Java**诊断工具**，专为生产环境设计，提供 **无需重启应用** 的动态诊断能力。其核心目标是帮助开发者快速定位线上问题

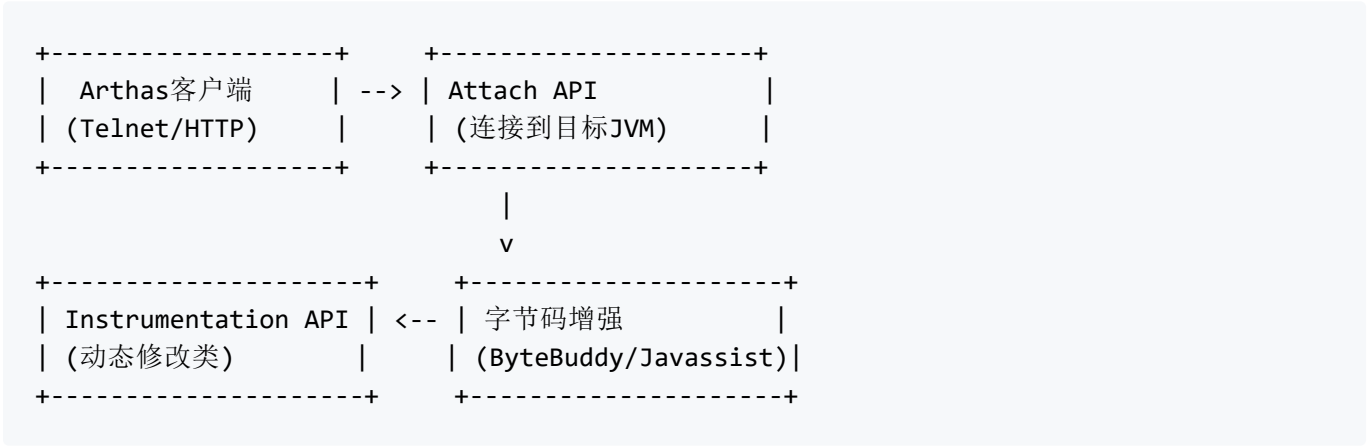
官网：https://arthas.aliyun.com/

2. 功能概述

功能类别	核心能力
监控类	方法调用监控、统计耗时、追踪调用链路
诊断类	查看JVM加载类信息、反编译代码、查看方法参数/返回值
热更新	动态修改已加载类的字节码（热修复）
线程分析	定位死锁、查看线程堆栈、监控线程状态
OGNL表达式	直接执行对象操作（获取Spring Bean、修改字段值等）
交互模式	命令行交互 + Web Console（3.5.0+）

3. 技术原理

3.1 核心架构



3.2 关键技术点

1. **Instrumentation API**
   - 通过 `java.lang.instrument` 实现动态字节码修改
   - 利用 `ClassFileTransformer` 接口插入诊断逻辑
2. **Attach机制**
   - 使用 `VirtualMachine.attach()` 注入Agent到目标JVM
   - 通过Socket通信实现客户端与服务端交互
3. **类加载隔离**
   - Arthas自身类由独立的 `ClassLoader` 加载
   - 避免与业务代码的类冲突
4. **异步数据采集**
   - 通过 `Advice` 机制在方法入口/出口插入监控代码
   - 数据通过环形缓冲区异步传输，降低性能损耗

4. 基础命令详解

4.1 高频核心命令

命令	参数示例	功能说明
`trace`	`trace com.example.Service * '#cost>100'`	追踪方法调用耗时（过滤>100ms的调用）
`watch`	`watch com.example.Service doSomething "{params,returnObj}" -x 3`	监控方法入参和返回值（展开3层对象）
`jad`	`jad com.example.Service`	反编译指定类的字节码
`redefine`	`redefine -c 327a2b4 /tmp/NewService.class`	热更新已加载的类
`thread`	`thread -n 3`	显示最忙的3个线程堆栈
`ognl`	`ognl '@com.example.Bean@FIELD'`	执行OGNL表达式获取/修改对象
`dashboard`	`dashboard -i 2000`	实时监控面板（2秒刷新间隔）

4.2 使用实践

性能瓶颈排查

场景：某接口响应时间突然变长

```
# 1. 定位高耗时方法
trace com.example.UserService get* '#cost>200' -n 3

# 2. 查看方法参数/返回值
watch com.example.UserService getUserById params[0] -x 2

# 3. 分析调用链路
stack com.example.UserService getUserById
```

输出示例：

```
method=getUserById cost=356ms
    +---[0.345ms] java.sql.Connection.prepareStatement()
    +---[354ms] java.sql.ResultSet.next()  // 数据库查询耗时过高
```

动态修改日志级别

```
# 查看当前日志配置
logger --name ROOT

# 动态调整日志级别
logger --name ROOT --level debug
```

热修复代码

使用`jad`反编译类：

```
jad --source-only com.example.BugService > /tmp/BugService.java
```

修改代码后编译：

```
mc /tmp/BugService.java -d /tmp
```

热加载新类：

```
redefine /tmp/com/example/BugService.class
```

vmtool

vmtool 是 Arthas 3.6+ 版本引入的 **对象操作工具**，支持直接操作 JVM 堆内存中的对象，典型场景包括：

- 动态获取/修改对象属性值
- 执行对象方法（包括私有方法）
- 查找特定类的所有实例
- 与 Spring 等框架深度集成（如获取 Bean）

场景1：获取 Spring 上下文中的 Bean

```
vmtool --action getInstances --classLoaderClass
org.springframework.boot.loader.LaunchedURLClassLoader --className
org.springframework.context.ApplicationContext --express 'instances[0]'
```

```
$ vmtool --action getInstances --className java.lang.String --limit 10
@String[][
    @String[com/taobao/arthas/core/shell/session/Session],
    @String[com.taobao.arthas.core.shell.session.Session],
    @String[com/taobao/arthas/core/shell/session/Session],
    @String[com/taobao/arthas/core/shell/session/Session],
    @String[com/taobao/arthas/core/shell/session/Session.class],
    @String[com/taobao/arthas/core/shell/session/Session.class],
    @String[com/taobao/arthas/core/shell/session/Session.class],
    @String[com/],
    @String[java/util/concurrent/ConcurrentHashMap$ValueIterator],
    @String[java/util/concurrent/locks/LockSupport],
]
```

> 通过 `--limit` 参数，可以限制返回值数量，避免获取超大数据时对 JVM 造成压力。默认值是 10。

场景2：动态修改配置参数

不重启服务调整缓存阈值

```
# 1. 查找配置类实例
vmtool --action getInstances --className com.example.CacheConfig

# 2. 修改 maxSize 字
vmtool --action getInstances --className com.example.CacheConfinfig --express
'instances[0].maxSize=500'
```

场景3：强制GC

```
vmtool --action forceGc
```

可以结合 `vmoption` 命令动态打开 `PrintGC` 开关

vmoption

```
# 查看所有option
vmoption

# 开启option
vmoption PrintGC true
```

retransform

retransform 是 Arthas 中用于 **重新转换已加载类** 的命令，与 `redefine` 命令相比，其核心差异在于：

- 通过 JVM `Instrumentation#retransformClasses` 机制触发类重新加载
- 支持 **多次修改生效**（`redefine` 存在单类多次修改失效问题）
- 适用于需要 **动态增强字节码** 的场景（如添加日志、监控埋点）

场景：动态修复方法逻辑

步骤1：反编译目标类

```
jad --source-only com.example.DataProcessor > arthas-output/DataProcessor.java
```

步骤2：修改代码

```
// 原始代码
public void process(String data) {
    // 存在空指针风险的代码
    System.out.println(data.toLowerCase());
}

// 修改后代码（增加空判断）
public void process(String data) {
    if (data != null) {
        System.out.println(data.toLowerCase());
    } else {
        System.out.println("[WARN] Null data received");
    }
}
```

步骤3：编译修改后的类

```
# 编译修改后的Java文件
mc arthas-output/DataProcessor.java -d arthas-output
```

步骤4：执行热更新

```
# 加载修改后的字节码
retransform arthas-output/com/example/DataProcessor.class

# 验证加载结果（显示 retransform entry）
retransform -l
```

或者

```
redefine arthas-output/com/example/DataProcessor.class
```

```
# 清除指定转换记录
retransform -d 1

# 完成操作后建议重置
retransform --resetAll
```

消除 retransform 的影响

如果对某个类执行 retransform 之后，想消除影响，则需要：

- 删除这个类对应的 retransform entry

  ```
  retransform -d 1
  retransform --deleteAll
  ```

- 重新触发 retransform

  ```
  retransform --classPattern demo.MathGame
  ```

> 如果不清除掉所有的 retransform entry，并重新触发 retransform ，则 arthas stop 时，retransform 过的类仍然生效。

retransform 与 redefine 的对比

特性	retransform	redefine
底层机制	调用 Instrumentation API	直接替换类定义
多次修改支持	✅ 支持	❌ 单类仅能生效一次
字节码增强	可与ClassFileTransformer配合	仅替换原始字节码
适用场景	动态添加监控/日志	紧急修复业务逻辑

thread

`thread` 是 Arthas 中用于 **线程状态分析** 的核心命令，支持以下关键操作：

参数	功能说明	示例
`thread`	显示所有线程的统计信息（状态、CPU 占用率等）	`thread`
`thread -b`	**检测死锁**，列出阻塞其他线程的线程（直接显示死锁链）	`thread -b` 35
`thread ID`	查看指定线程的完整堆栈信息	`thread 58`
`thread -n 3`	显示 CPU 占用率最高的前 N 个线程（用于快速定位性能瓶颈）	`thread -n 5` 7
`thread --state BLOCKED`	过滤特定状态的线程（如 `BLOCKED` / `WAITING`）	`thread --state BLOCKED`

利用 Arthas 排查死锁的步骤

```
[arthas@11596]$ thread -b

"t1" Id=10 BLOCKED on java.lang.Object@26dee7d7 owned by "t2" Id=11
    at test.Deadlock.lambda$main$0(Deadlock.java:24)
    -  blocked on java.lang.Object@26dee7d7
    -  locked java.lang.Object@13a631ce <---- but blocks 1 other threads!
```

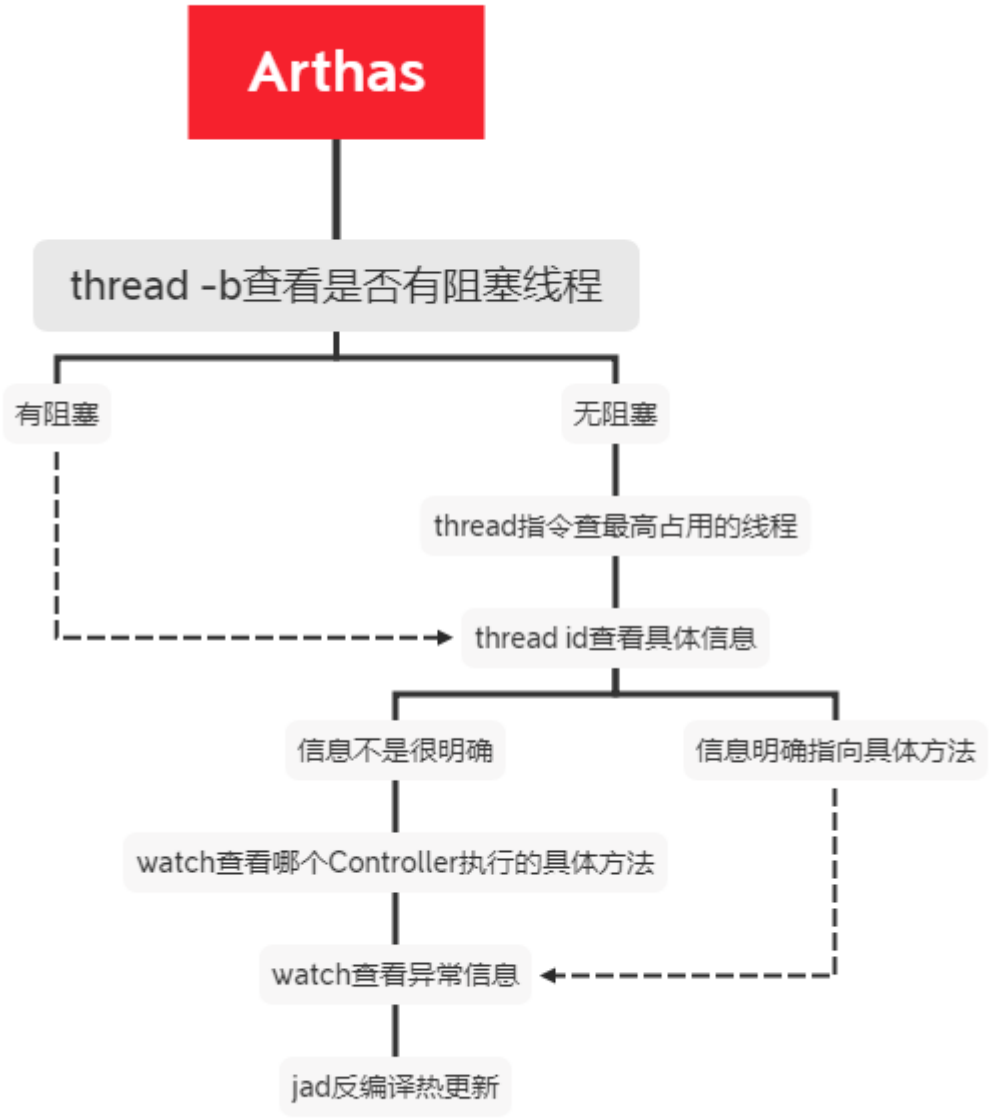
- **输出解读**：
  - 直接显示 **死锁线程 ID**（如 `t1` 和 `t2`）及其持有的锁对象 35。
  - 标识 `locked` 和 `blocked` 的锁对象，明确锁竞争关系。

查看具体线程堆栈

```
# 查看线程 t1 的堆栈
thread 10

# 输出示例：
"t1" Id=10 BLOCKED
    at com.example.Deadlock.lambda$main$0(Deadlock.java:24)
    -  locked <0x000000076ac710a0> (java.lang.Object)
    -  blocked on <0x000000076ac710b0> (java.lang.Object)
```

- **关键信息**：
  - 线程当前状态（如 `BLOCKED`）。
  - 已持有的锁（`locked`）和等待的锁（`blocked on`）。



watch

watch 是 Arthas 中使用频率最高的 **方法观测命令**，可实时捕获方法的：

- 方法入参（支持参数索引定位）
- 返回值/异常对象
- 方法执行耗时
- 上下文环境变量

```
watch 类全限定名 方法名 观察表达式 [条件过滤] [参数选项]
```

参数	说明
`-b`	观察方法调用前（Before事件）
`-e`	观察方法异常返回（Exception事件）
`-s`	观察方法正常返回（Success事件）
`-f`	观察方法结束（Finish事件，包含成功/异常）
`-x`	对象展开层级（默认1层，最大值4）
`-n`	执行次数限制
`#cost`	耗时过滤（单位ms）
`params`	表示方法所有参数数组
`returnObj`	方法返回对象
`target`	this 对象
`throwExp`	异常对象

- watch 命令定义了 4 个观察事件点，即 `-b` 函数调用前，`-e` 函数异常后，`-s` 函数返回后，`-f` 函数结束后
- 4 个观察事件点 `-b`、`-e`、`-s` 默认关闭，`-f` 默认打开，当指定观察点被打开后，在相应事件点会对观察表达式进行求值并输出
- 这里要注意 `函数入参` 和 `函数出参` 的区别，有可能在中间被修改导致前后不一致，除了 `-b` 事件点 `params` 代表函数入参外，其余事件都代表函数出参
- 当使用 `-b` 时，由于观察事件点是在函数调用前，此时返回值或异常均不存在
- 在 watch 命令的结果里，会打印出 `location` 信息。`location` 有三种可能值：`AtEnter`，`AtExit`，`AtExceptionExit`。对应函数入口，函数正常 return，函数抛出异常。

```
# 监控第二个参数的值（索引从0开始）
$ watch com.example.OrderService createOrder params[1] -x 3

# 输出示例：
method=createOrder location=AtEnter
@OrderItem[
    productId=@Integer[1005],
    amount=@Integer[2],
    price=@Double[199.99]
]

# 观察函数调用入口的参数和返回值
$ watch demo.MathGame primeFactors "{params,returnObj}" -x 2 -b
Press Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 50 ms.
ts=2018-12-03 19:23:23; [cost=0.0353ms] result=@ArrayList[
    @Object[][
        @Integer[-1077465243],
    ],
    null,
]

# 同时观察函数调用前和函数返回后
$ watch demo.MathGame primeFactors "{params,target,returnObj}" -x 2 -b -s -n 2
Press Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 46 ms.
ts=2018-12-03 19:29:54; [cost=0.01696ms] result=@ArrayList[
    @Object[][
        @Integer[1],
    ],
    @MathGame[
        random=@Random[java.util.Random@522b408a],
        illegalArgumentCount=@Integer[13038],
    ],
    null,
]
ts=2018-12-03 19:29:54; [cost=4.277392ms] result=@ArrayList[
    @Object[][
        @Integer[1],
    ],
    @MathGame[
        random=@Random[java.util.Random@522b408a],
        illegalArgumentCount=@Integer[13038],
    ],
    @ArrayList[
        @Integer[2],
        @Integer[2],
        @Integer[2],
        @Integer[5],
        @Integer[5],
```

```
        @Integer[73],
        @Integer[241],
        @Integer[439],
    ],
]

# 条件表达式的例子
$ watch demo.MathGame primeFactors "{params[0],target}" "params[0]<0"
Press Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 68 ms.
ts=2018-12-03 19:36:04; [cost=0.530255ms] result=@ArrayList[
    @Integer[-18178089],
    @MathGame[demo.MathGame@41cf53f9],
]

# 按照耗时进行过滤
$ watch demo.MathGame primeFactors '{params,returnObj}' '#cost>200' -x 2
Press Ctrl+C to abort.
Affect(class-cnt:1 , method-cnt:1) cost in 66 ms.
ts=2018-12-03 19:40:28; [cost=2112.168897ms] result=@ArrayList[
    @Object[][
        @Integer[1],
    ],
    @ArrayList[
        @Integer[5],
        @Integer[428379493],
    ],
]

# json序列化输出对象
$ watch com.xx.xx.XXXservice findById
'@com.xx.utils.JsonUtils@object2String(returnObj)' -s -x 1


# 预防大对象序列化
$ watch com.xx.xx.XXXservice findById
'returnObj != null && returnObj.size() < 1000 ?
com.xx.utils.JsonUtils@object2String(returnObj) : "too_large"' -s
```

tt

> 方法执行数据的时空隧道，记录下指定方法每次调用的入参和返回信息，并能对这些不同的时间下调用进行观测

TimeTunnel（**时间隧道**） 是 Arthas 中用于 **记录和回放方法调用** 的高级诊断工具，主要解决以下问题：

- **现场复现**：捕获特定方法的历史调用参数/返回值，无需重新触发请求

- **问题定位**：分析异常调用链路的上下文环境
- **动态调试**：修改入参后重新执行方法逻辑（沙箱环境）

记录方法调用

```
# 基本记录模式
tt -t com.example.service.UserService queryUser -n 5

# 参数说明：
# -t ： 开始记录方法调用
# 类名.方法名 ： 监控的目标方法
# -n ： 最多记录次数（防止内存溢出）


 INDEX   TIMESTAMP           COST(ms)  IS-RET  IS-EXP   OBJECT       CLASS
METHOD
 1000    2024-05-06 14:30:45 12        true    false    0x3d4e5a6    UserService
queryUser
```

查看记录详情

```
# 查看所有记录
tt -l

# 查看指定索引的调用详情
tt -i 1000

INDEX            1003
 GMT-CREATE       2018-12-04 11:15:41
 COST(ms)         0.186073
 OBJECT           0x4b67cf4d
 CLASS            demo.MathGame
 METHOD           primeFactors
 IS-RETURN        false
 IS-EXCEPTION     true
 PARAMETERS[0]    @Integer[-564322413]
 THROW-EXCEPTION  java.lang.IllegalArgumentException: number is: -564322413, need >= 2
                      at demo.MathGame.primeFactors(MathGame.java:46)
                      at demo.MathGame.run(MathGame.java:24)
                      at demo.MathGame.main(MathGame.java:16)

Affect(row-cnt:1) cost in 11 ms.

# 显示参数和返回值（-w 使用OGNL表达式）
tt -i 1000 -w 'params[0]'
tt -i 1000 -w 'returnObj'
```

方法调用回放

```
# 回放指定记录（重新执行方法）
tt -i 1000 -p

# 回放时修改参数
tt -i 1000 -p --params "newParam1,newParam2"

# 回放时打印堆栈
tt -i 1000 -p --replay-stack
```

注意事项

- tt 命令的实现是：把函数的入参/返回值等，保存到一个 `Map<Integer, TimeFragment>` 里，默认的大小是 100。
- tt 相关功能在使用完之后，需要手动释放内存，否则长时间可能导致OOM。退出 arthas 不会自动清除 tt 的缓存 map

```
# 通过索引删除指定的 tt 记录
tt -d 1001
# 清除所有的 tt 记录
tt --delete-all
```

- 需要强调的点

    1. **ThreadLocal 信息丢失**

        很多框架偷偷的将一些环境变量信息塞到了发起调用线程的 ThreadLocal 中，由于调用线程发生了变化，这些 ThreadLocal 线程信息无法通过 Arthas 保存，所以这些信息将会丢失。

        一些常见的 CASE 比如：鹰眼的 TraceId 等。

    2. **引用的对象**

        需要强调的是，`tt` 命令是将当前环境的对象引用保存起来，但仅仅也只能保存一个引用而已。如果方法内部对入参进行了变更，或者返回的对象经过了后续的处理，那么在 `tt` 查看的时候将无法看到当时最准确的值。这也是为什么 `watch` 命令存在的意义。

实用实例

问题现场复现

```
# 1. 开始记录
tt -t com.example.service.UserService queryUser -n 10

# 2. 当问题复现后，查找异常记录 表达式检索 找到录制的index
tt -s 'isExp == true'

# 3. 分析异常参数
tt -i 1001 -w 'params[0]'
```

动态调试参数

```
# 1. 找到正常调用记录
tt -l

# 2. 修改参数后回放（如测试userId=0的边界情况）
tt -i 1002 -p --params "0"
```

验证修复方案

```
# 1. 热修复代码
jad com.example.service.UserService > /tmp/UserService.java
mc /tmp/UserService.java -d /tmp
redefine /tmp/com/example/service/UserService.class

# 2. 回放历史问题请求
tt -i 1003 -p
```