

19 | Java & C ++：代码级监控及常用计数器解析（上）

2020-02-03 高楼

性能测试实战30讲

[进入课程 >](#)



讲述：高楼

时长 13:03 大小 11.96M

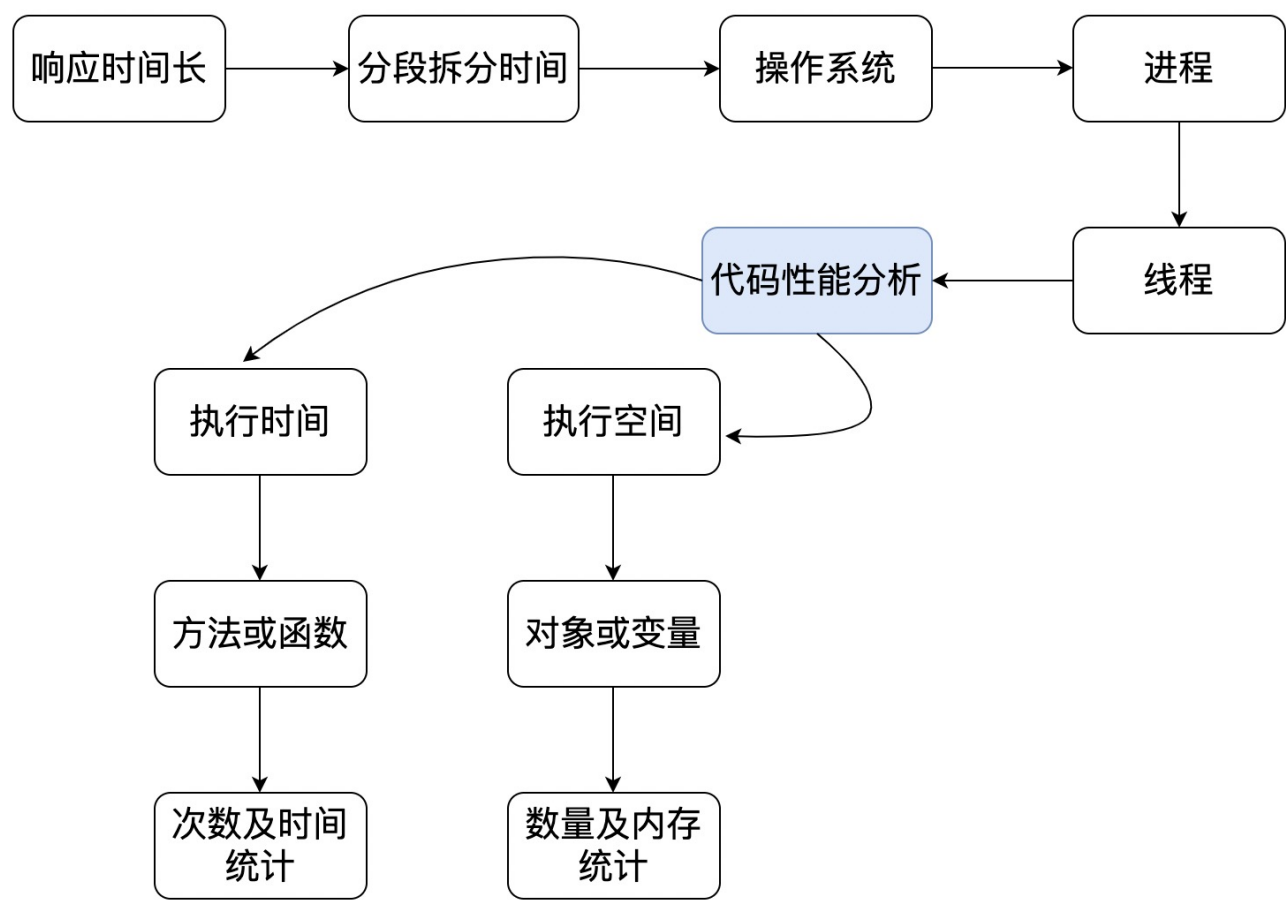


在性能测试分析中，有一部分人存在着一个思路上的误解，那就是一开始就一头扎进代码里，折腾代码性能。这是我非常反对的一种做法。

事实上，要想这么做，有一个前提，那就是架构中的其他组件都经过了千锤百炼，出现问题的可能性极低。

实际上，我凭着十几年的经验来看，大部分时候，代码出现严重性能瓶颈的情况还真是不多。再加上现在成熟的框架那么多，程序员们很多情况下只写业务实现。在这种情况下，代码出现性能瓶颈的可能性就更低了。

但我们今天终归要说代码级的监控及常用的计数器。如何去评估一个业务系统的代码性能呢？在我看来，分析的思路是下面这个样子的。




从上图可以看到，分析的时候有两个关键点：执行时间和执行空间。我相信很多人都清楚，我们要很快找到执行时间耗在哪一段和空间耗在哪里。

现在我们来实际操作一下，看如何判断。

Java 类应用查找方法执行时间

首先你得选择一个合适的监控工具。Java 方法类的监控工具有很多，这里我选择 JDK 里自带的 jvisualvm。

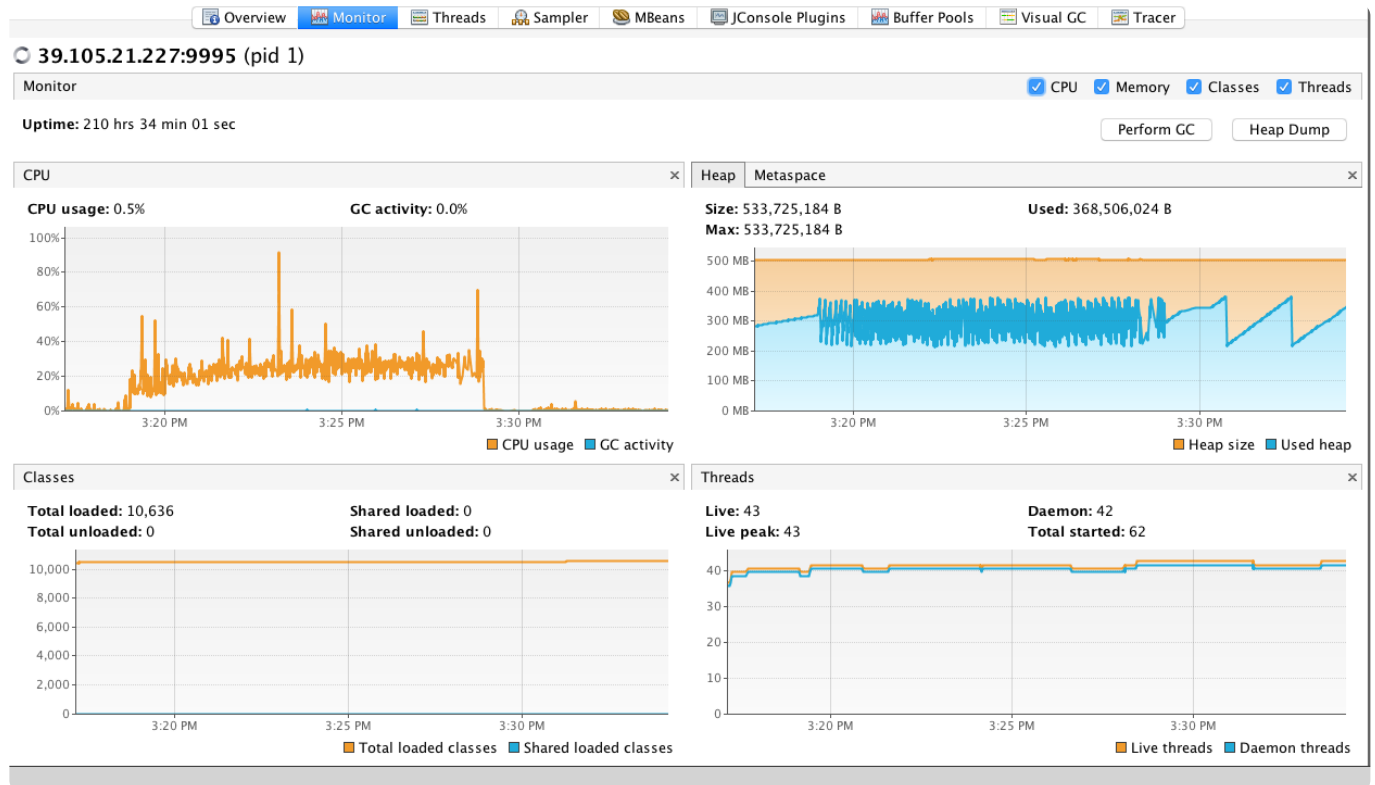
顺便说一下，我的 Java 版本号是这个：

 复制代码

```
1 (base) GaoLouMac:~ Zee$ java -version
2 java version "1.8.0_111"
```

- 3 Java(TM) SE Runtime Environment (build 1.8.0_111-b14)
- 4 Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode)

打开应用服务器上的 JMX 之后，连上 jvisualvm，你会看到这样的视图。



这里再啰嗦一下我们的目标，这时我们要找到消耗 CPU 的方法，所以要先点Sampler - CPU，你可以看到如下视图。

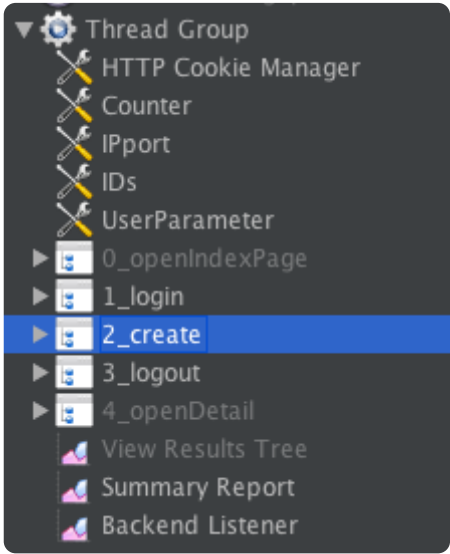
Hot Spots - Method	Self Time (%) ▾	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)	Samples
org.apache.tomcat.util.threads.TaskQueue.take ()	<div></div>	22,457,882 ms (56.6%)	559 ms	22,457,882 ms	559 ms	5,742
com.mysql.jdbc.util.ReadAheadInputStream.fill ()	<div></div>	6,964,805 ms (17.6%)	6,964,805 ms	6,964,805 ms	6,964,805 ms	4,208
org.apache.catalina.core.ContainerBase\$ContainerBackgroundProcessor.run ()	<div></div>	5,598,203 ms (14.1%)	0.000 ms	5,598,635 ms	431 ms	1
org.apache.tomcat.util.net.NioEndpoint\$Poller.run ()	<div></div>	3,908,486 ms (9.9%)	5,040 ms	3,915,917 ms	12,471 ms	4
redis.clients.util.RedisInputStream.ensureFill ()	<div></div>	310,735 ms (0.8%)	310,735 ms	310,735 ms	310,735 ms	302
com.jt.blog.common.validate.ValidateCode.write ()	<div></div>	153,878 ms (0.4%)	153,878 ms	157,703 ms	157,703 ms	145
org.apache.tomcat.util.net.NioChannel.read ()	<div></div>	24,709 ms (0.1%)	24,709 ms	24,709 ms	24,709 ms	24
redis.clients.util.RedisOutputStream.flushBuffer ()	<div></div>	23,798 ms (0.1%)	23,798 ms	23,798 ms	23,798 ms	27
com.mysql.jdbc.util.ReadAheadInputStream.available ()	<div></div>	22,531 ms (0.1%)	22,531 ms	22,531 ms	22,531 ms	20
org.apache.tomcat.util.net.NioChannel.write ()	<div></div>	20,185 ms (0.1%)	20,185 ms	20,185 ms	20,185 ms	19
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.resolveFieldReferences ()	<div></div>	14,287 ms (0%)	14,287 ms	14,287 ms	14,287 ms	15
org.springframework.cache.interceptor.CacheOperationInterceptor.getOperation ()	<div></div>	10,924 ms (0%)	10,924 ms	10,924 ms	10,924 ms	13
org.springframework.web.context.request.async.WebAsyncManagerImpl.registerStatement ()	<div></div>	8,013 ms (0%)	8,013 ms	8,013 ms	8,013 ms	9
com.mysql.jdbc.ConnectionImpl.registerStatement ()	<div></div>	7,085 ms (0%)	7,085 ms	7,085 ms	7,085 ms	7
com.github.pagehelper.SqlUtil.getParamValue ()	<div></div>	6,923 ms (0%)	6,923 ms	7,646 ms	7,646 ms	8
org.apache.tomcat.util.threads.TaskQueue.offer ()	<div></div>	6,641 ms (0%)	6,641 ms	6,641 ms	6,641 ms	14
org.apache.ibatis.io.ClassLoaderWrapper.classForName ()	<div></div>	5,884 ms (0%)	5,884 ms	5,884 ms	5,884 ms	12
org.springframework.core.annotation.AnnotationMethodReturnValueHandler.handle ()	<div></div>	5,241 ms (0%)	5,241 ms	5,915 ms	5,915 ms	17
freemarker.ext.beans.BeanWrapper.invokeMethod ()	<div></div>	5,026 ms (0%)	5,026 ms	5,700 ms	5,700 ms	5
org.apache.ibatis.reflection.TypeParameterResolver.resolveFieldTypes ()	<div></div>	4,429 ms (0%)	4,429 ms	4,429 ms	4,429 ms	4
org.apache.ibatis.plugin.Plugin.getSignatureMap ()	<div></div>	4,070 ms (0%)	4,070 ms	4,070 ms	4,070 ms	4
freemarker.core.FMParserTokenManager.jjMoveNfa_0 ()	<div></div>	3,469 ms (0%)	3,469 ms	3,469 ms	3,469 ms	1
com.mysql.jdbc.StatementImpl.removeOpenResultSet ()	<div></div>	3,337 ms (0%)	3,337 ms	3,337 ms	3,337 ms	3
org.apache.tomcat.util.net.NioEndpoint\$Poller.addEvent ()	<div></div>	3,328 ms (0%)	3,328 ms	3,328 ms	3,328 ms	3
com.mysql.jdbc.MysqlIO.send ()	<div></div>	3,218 ms (0%)	3,218 ms	3,218 ms	3,218 ms	4
freemarker.core.Environment.visit ()	<div></div>	2,989 ms (0%)	2,989 ms	20,824 ms	20,824 ms	71
com.fasterxml.jackson.databind.introspect.AnnotatedClass.findMethodsWithName ()	<div></div>	2,897 ms (0%)	2,897 ms	2,897 ms	2,897 ms	3
org.apache.ibatis.reflection.TypeParameterResolver.resolveReturnTypes ()	<div></div>	2,742 ms (0%)	2,742 ms	3,277 ms	3,277 ms	3
org.springframework.core.annotation.AnnotationMethodReturnValueHandler.handle ()	<div></div>	2,556 ms (0%)	2,556 ms	2,556 ms	2,556 ms	2
org.springframework.web.method.support.HandlerMethodReturnValueHandler.handle ()	<div></div>	2,268 ms (0%)	2,268 ms	2,268 ms	2,268 ms	2
org.apache.catalina.webresources.AbstractArchiveResourceSet.open ()	<div></div>	2,084 ms (0%)	2,084 ms	2,084 ms	2,084 ms	2
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.resolveFieldReferences ()	<div></div>	2,029 ms (0%)	2,029 ms	25,811 ms	25,811 ms	28
org.apache.catalina.webresources.AbstractSingleArchiveResourceSet.open ()	<div></div>	1,998 ms (0%)	1,998 ms	4,083 ms	4,083 ms	4
org.apache.ibatis.executor.resultset.DefaultResultSetHandler.populateResultSet ()	<div></div>	1,914 ms (0%)	1,914 ms	8,127 ms	8,127 ms	8
org.springframework.core.annotation.AnnotationMethodReturnValueHandler.handle ()	<div></div>	1,873 ms (0%)	1,873 ms	2,824 ms	2,824 ms	12
org.springframework.util.ClassUtils.isAssignable ()	<div></div>	1,871 ms (0%)	1,871 ms	1,871 ms	1,871 ms	2

从上图可以看到方法执行的累积时间，分别为自用时间百分比、自用时间、自用时间中消耗 CPU 的时间、总时间、总时间中消耗 CPU 的时间、样本数。

从这些数据中就可以看到方法的执行效率了。

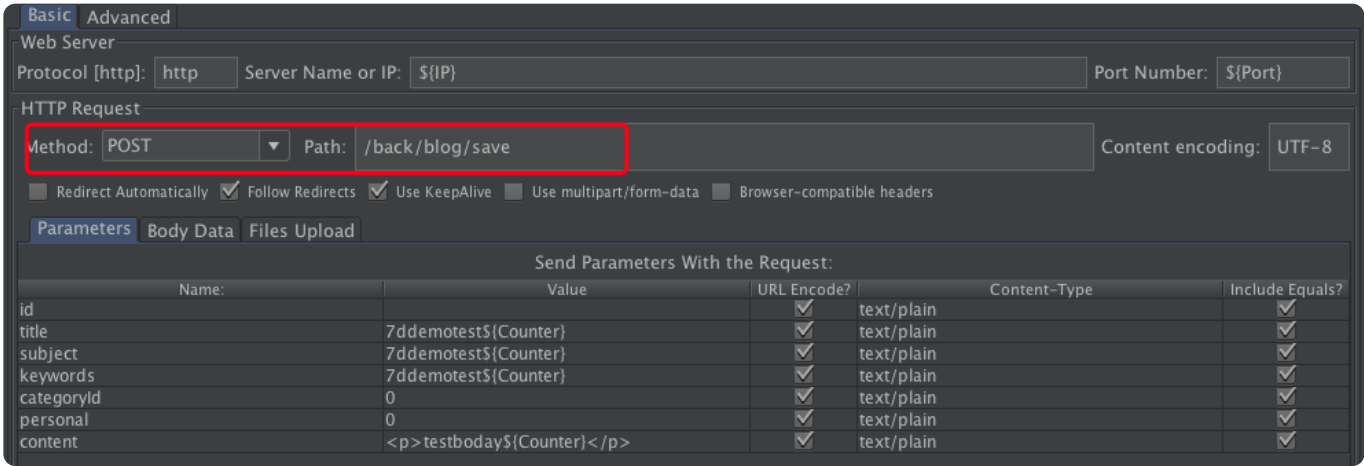
但是，这里面 Method 这么多，我怎么知道哪个跟我的方法执行时间有关呢？比如说上面这个应用中，最消耗 CPU 的是 JDBC 的一个方法 fill。这合理吗？

先来看一下我的脚本。



从结构上你就能看出来，我做了登录，然后就做了创建的动作，接着就退出了。

这几个操作和数据库都有交互。拿 create 这个步骤来说，它的脚本非常直接，就是一个 POST 接口。



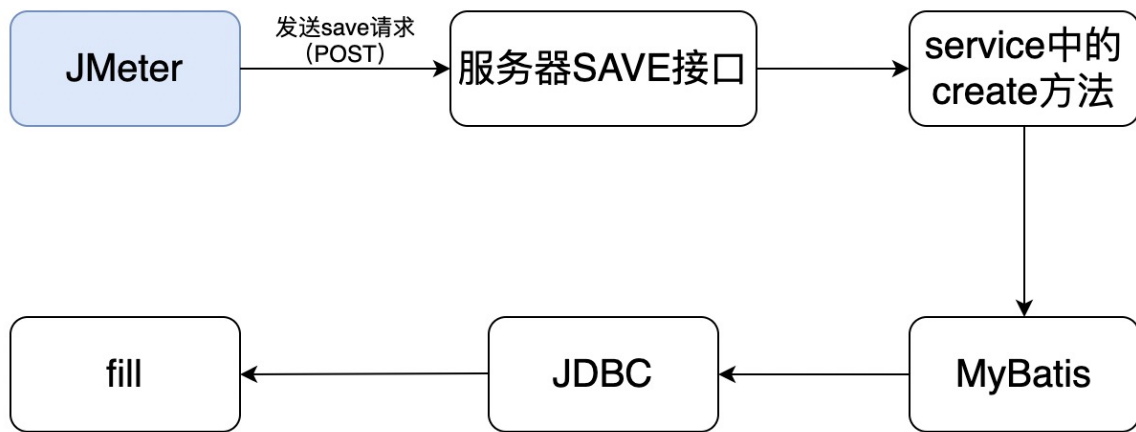
还记得前面我们怎么说查看后端的运行逻辑的吗？后端接收这个 POST 的代码如下：


```
1  @RequestMapping("/save")
2  @ResponseBody
3  public Object save(Blog blog, HttpSession session){
4      try{
5          Long id = blog.getId();
6          if(id==null){
7              User user = (User)session.getAttribute("user");
8              blog.setAuthor(user.getName());
9              blog.setUserId(user.getId());
10             blog.setCreateTime(new Date());
11             blog.setLastModifyTime(new Date());
12             blogWriteService.create(blog);
13         }else {
14             blog.setLastModifyTime(new Date());
15             blogWriteService.update(blog);
16         }
17     }catch (Exception e){
18         throw new JsonResponseException(e.getMessage());
19     }
20     return true;
21 }
```



这段代码的功能就是讲前端内容接收过来放到实体中，然后通过 create 方法写到数据库中。那么 create 是怎么实现的呢？

```
1  public void create(Blog blog) {
2      mapper.insert(blog);
3      BlogStatistics blogStatistics = new BlogStatistics(blog.getId());
4      blogStatisticsMapper.insert(blogStatistics);
}
```

它就是一个 mapper.insert，显然这个 create 是我们自己实现的代码，里面其实没有什么逻辑。而 ReadAheadInputStream.fill 是 create 中的 MyBatis 调用的 JDBC 中的方法。从压力工具到数据库的调用逻辑就是：








而我们看到的最耗时的方法是最后一个，也就是 fill。实际上，我们应该关心的是 save 接口到底怎么样。我们来过滤下看看。

CPU samples		Thread CPU Time						Thread Dump	
		Snapshot							
Hot Spots - Method		Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)	Samples		
com.controller.BackBlogController\$\$EnhancerBySpringCGLI			0.000 ms	(0%)	0.000 ms	945,417 ms	945,417 ms	839	
com.controller.BackBlogController.save ()			0.000 ms	(0%)	0.000 ms	945,417 ms	945,417 ms	839	





从 save 的结果上来看，它本身并没有耗什么时间，都是后面的调用在消耗时间。

我们再来看看 create。

CPU samples		Thread CPU Time							
<div><div></div><div> Snapshot</div></div>								Thread Dump	
Hot Spots - Method		Self Time [%] ▾	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)	Samples		
	com.service.BlogWriteServiceImpl.create ()		0.000 ... (0%)	0.000 ms	167,766 ms	167,766 ms	166		
	org.springframework.transaction.interceptor.TransactionAspectSupport.createTransactionIfNec		0.000 ... (0%)	0.000 ms	23,024 ms	23,024 ms	19		
	org.springframework.web.servlet.view.UrlBasedViewResolver.createView ()		0.000 ... (0%)	0.000 ms	18,509 ms	18,509 ms	17		
	org.springframework.web.servlet.view.AbstractCachingViewResolver.createView ()		0.000 ... (0%)	0.000 ms	18,509 ms	18,509 ms	17		
	org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.create		0.000 ... (0%)	0.000 ms	2,936 ms	2,936 ms	3		
	com.alibaba.druid.pool.DruidAbstractDataSource.createPhysicalConnection ()		0.000 ... (0%)	0.000 ms	395 ms	395 ms	2		
	com.fasterxml.jackson.databind.ser.BeanSerializerFactory.createSerializer ()		0.000 ... (0%)	0.000 ms	538 ms	538 ms	1		
	com.fasterxml.jackson.databind.SerializerProvider._createUntypedSerializer ()		0.000 ... (0%)	0.000 ms	538 ms	538 ms	1		
	com.fasterxml.jackson.databind.SerializerProvider._createAndCacheUntypedSerializer ()		0.000 ... (0%)	0.000 ms	538 ms	538 ms	1		
	org.springframework.web.method.HandlerMethod.createWithResolvedBean ()		0.000 ... (0%)	0.000 ms	678 ms	678 ms	1		
	com.alibaba.druid.filter.stat.StatFilter.createSqlStat ()		0.000 ... (0%)	0.000 ms	521 ms	521 ms	1		
	com.common.validate.ValidateCode.createCode ()		0.000 ... (0%)	0.000 ms	642 ms	642 ms	1		
	com.alibaba.druid.pool.DruidDataSource\$CreateConnectionThread.run ()		0.000 ... (0%)	0.000 ms	0.000 ms	0.000 ms	1		
	com.mysql.jdbc.ConnectionImpl.createNewIO ()		0.000 ... (0%)	0.000 ms	395 ms	395 ms	1		
	com.fasterxml.jackson.databind.ser.BeanSerializerFactory._createSerializer2 ()		0.000 ... (0%)	0.000 ms	538 ms	538 ms	1		


它本身也没消耗什么时间。

顺着逻辑图，我们再接着看 MyBatis 中的 insert 方法。

CPU samples		Thread CPU Time							
		 Snapshot						 Thread Dump	
Hot Spots - Method		Self Time [%] ▾	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)	Samples		
org.mybatis.spring.SqlSessionTemplate.insert ()			0.000 ... (0%)	0.000 ms	170,971 ms	170,971 ms	172		
org.apache.ibatis.session.defaults.DefaultSqlSession.insert ()			0.000 ... (0%)	0.000 ms	169,031 ms	169,031 ms	170		

就这样一层层找下去，最后肯定就找到了 fill 这个方法了。但是你怎么知道整个调用逻辑中有哪些层级呢？你说我可以看源码。当然不是不可以。但要是没有源码呢？做性能分析的人经常没有源码呀。

这个时候，我们就要来看栈了。这里我打印了一个调用栈，我们来看下这个逻辑。

 复制代码

```
1  "http-nio-8080-exec-1" - Thread t@42
2      java.lang.Thread.State: RUNNABLE
3      .....
4      at com.mysql.jdbc.util.ReadAheadInputStream.fill(ReadAheadInputStream.java:11)
5      .....
6      .....
7      at com.sun.proxy.$Proxy87.create(Unknown Source)
8      .....
9      at com.blog.controller.BackBlogController.save(BackBlogController.java:85)
10     .....
11     at java.lang.Thread.run(Thread.java:745)
12
13
14     Locked ownable synchronizers:
15     - locked <4b6968c3> (a java.util.concurrent.ThreadPoolExecutor$Worker)
```

我把其他的都给清掉了，我们只看最简单的栈逻辑，其中 UnknownSource 的部分是因为反射实现的 insert 没有把源码反编译出来。

其实这个栈有 117 行，我怕你看晕。

从这一层一层的关系中，我们就可以知道调用逻辑了。知道调用逻辑的方法有很多，看源码也行，看编译后运行的代码也行，关键在于知道谁调了谁，这样就行了。

我这个还算是清晰的调用逻辑，要是代码调用关系再复杂一些，分分钟有想死有没有？

不过比较好的是，像 jvisualvm 这样的工具给我们提供了很多便利。这时可能有人会跳起来了，为什么不用 Arthas、BTrace 之类的工具呢？如果你喜欢的话，可以把 Arthas 弄上，像下面这样。

 复制代码


```
1 [arthas@1]$ trace com.blog.controller.BackBlogController save
```

```

2 Press Q or Ctrl+C to abort.
3 Affect(class-cnt:2 , method-cnt:2) cost in 320 ms.
4 `---ts=2020-01-06 10:38:37;thread_name=http-nio-8080-exec-2;id=2b;is_daemon=tru
5     `---[29.048684ms] com.blog.controller.BackBlogController$$EnhancerBySpring
6         `---[28.914387ms] org.springframework.cglib.proxy.MethodInterceptor:in
7             `---[27.897315ms] com.blog.controller.BackBlogController:save()
8                 .....
9                 `---[24.192784ms] com.blog.service.BlogWriteService:create() #i

```

这也能看出来 creat 是消耗了时间的。如果你接着跟踪 create 方法。如下所示：


 复制代码

```

1 [arthas@1]$ trace com.blog.service.BlogWriteService create //这一行是arthas中跟踪
2 Press Q or Ctrl+C to abort.
3 Affect(class-cnt:2 , method-cnt:2) cost in 199 ms. //被跟踪方法的处理次数和时长
4 `---ts=2020-01-06 10:41:51;thread_name=http-nio-8080-exec-4;id=2f;is_daemon=tru
5     `---[6.939189ms] com.sun.proxy.$Proxy87:create()
6     `---ts=2020-01-06 10:41:51;thread_name=http-nio-8080-exec-10;id=38;is_daemon=t
7         `---[4.144799ms] com.blog.service.BlogWriteServiceImpl:create() //写接
8             +---[2.131934ms] tk.mybatis.mapper.common.Mapper:insert() #24 //i
9                 .....
10                `---[1.95441ms] com.blog.mapper.BlogStatisticsMapper:insert() #26

```

要是接着往下跟踪，就可以看到反射这一块了。

 复制代码

```

1 [arthas@1]$ trace tk.mybatis.mapper.common.Mapper insert
2 Press Q or Ctrl+C to abort.
3 Affect(class-cnt:5 , method-cnt:5) cost in 397 ms.
4 `---ts=2020-01-06 10:44:01;thread_name=http-nio-8080-exec-5;id=33;is_daemon=tru
5     `---[3.800107ms] com.sun.proxy.$Proxy80:insert()

```

类似的，你还可以玩 JDK 自带的工具 jdb，它也可以直接 attach 到一个进程上，调试能力也是不弱的。

在我看来，这些工具、手段都是为了实现从响应时间长<->代码行的分析过程。思路是最重要的。

另外也要说一下，现在有的 APM 工具也可以实现这样的功能，但是呢，我并不建议一开始就上这么细致的工具，因为不管 APM 产品吹得有多牛逼，它都是要消耗 10% 左右的 CPU 的。并且，你觉得直接在生产上装一个 APM 工具的 agent 到业务系统中是合理的吗？如果是自己实现的 metrics 方法，输出性能数据尚可接受，如果是别人的这类工具，还是算了。

在大部分时候，我都不建议在生产上用 APM 工具。万一生产上真的有极端的情况，需要看细致的性能问题，再临时 attach 上去，也可以做到。何必为了可能出现的问题而长时间地消耗资源呢。

总结

大部分时间里，性能测试和分析都在和时间打交道，而在时间的拆分逻辑中，我们在前面也提到过思路，如何一步步把时间拆解到应用当中，那就是**分段**。

当拆解到应用当中之后，就是抓函数方法的执行时间了。这是保证我们从前到后分析逻辑的关键一环，请你注意，是关键一环，而不是最初的一环。

通过这篇文章我想告诉你，在大部分的开发语言中，都有手段直接将方法的执行时间消耗抓出来，你可能现在还不知道是什么方法，没关系，因为跟踪的手段有很多，你可以临时去学习如何操作。

我只要在你的脑子里种下这样的一种印象，那就是，有办法抓到函数方法的执行时间消耗在哪里！

思考题

最后给你留两道思考题吧。我为什么不建议在生产环境中一开始就上 APM 类工具来抓取方法的执行时间呢？你有什么方法可以抓取到 Java 语言中的方法执行时间？如果你擅长其他语言，也可以描述其他语言中的方法执行时间抓取工具。

欢迎你在评论区写下你的思考，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

点击查看 

打卡学习，成为真正的性能测试高手



PC端用户扫码参与



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 18 | CentOS：操作系统级监控及常用计数器解析（下）

下一篇 20 | Java & C++：代码级监控及常用计数器解析（下）

精选留言 (1)

 写留言



嘟嘟爱学习

2020-02-08

我觉得某些生产环境还是可以直接上APM的：

1. 能接受10%性能损耗的，比如原来耗时1秒，上了变成1.1秒其实感觉不明显；原来高峰期CPU使用率30%，上了变成40%也还在可接受范围内；
2. APM的成功失败不影响业务的运行，就是即使APM挂了，业务也还能正常运行；
3. 在docker+k8s且又有大量虚拟机大量服务的情况下，上APM也是一个方案，不然当出现...

展开 ∨

作者回复：很不幸的是，你说的非常对。

我觉得我们对大量服务的场景其实需要的只是一个链路监控系统，这个功能APM基本都有提供，我们要用的就是这个功能而已。

另外，我不知道你有没有遇到过APM的agent导致业务系统挂掉的情况，在我的工作中有遇到过，一级故障，损失也是惨重。

所以用不用APM，只有在具体的应用场景中，测试好了再决定上不上吧。

