20 SpringBoot 服务性能优化

在开始对 SpringBoot 服务进行性能优化之前，你需要做一些准备，把 SpringBoot 服务的一些数据暴露出来。比如，你的服务用到了缓存，就需要把缓存命中率这些数据进行收集；用到了数据库连接池，就需要把连接池的参数给暴露出来。

我们这里采用的监控工具是 Prometheus，它是一个是时序数据库，能够存储我们的指标。SpringBoot 可以非常方便地接入到 Prometheus 中。

SpringBoot 如何开启监控？

创建一个 SpringBoot 项目后，首先加入 maven 依赖。

```
<dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-actuator</artifactId>
 </dependency>
 <dependency>
     <groupId>io.micrometer</groupId>
     <artifactId>micrometer-registry-prometheus</artifactId>
 </dependency>
 <dependency>
     <groupId>io.micrometer</groupId>
     <artifactId>micrometer-core</artifactId>
 </dependency>
```

然后，我们需要在 application.properties 配置文件中，开放相关的监控接口。

```
management.endpoint.metrics.enabled=true
management.endpoints.web.exposure.include=*
management.endpoint.prometheus.enabled=true
management.metrics.export.prometheus.enabled=true
```

启动之后，我们就可以通过访问监控接口来获取监控数据。

```
# HELP jvm_memory_committed_bytes The amount of memory in bytes that is committed for the Java virtual mach
# TYPE jvm_memory_committed_bytes gauge
jvm_memory_committed_bytes{area="nonheap",id="CodeHeap 'profiled nmethods'",} 9961472.0
jvm_memory_committed_bytes{area="heap",id="G1 Survivor Space",} 4194304.0
jvm_memory_committed_bytes{area="heap",id="G1 Old Gen",} 2.8311552E7
jvm_memory_committed_bytes{area="nonheap",id="Metaspace",} 4.233216E7
jvm_memory_committed_bytes{area="nonheap",id="CodeHeap 'non-nmethods'",} 2555904.0
jvm_memory_committed_bytes{area="heap",id="G1 Eden Space",} 2.7262976E7
jvm_memory_committed_bytes{area="nonheap",id="Compressed Class Space",} 5898240.0
jvm_memory_committed_bytes{area="nonheap",id="CodeHeap 'non-profiled nmethods'",} 2555904.0
# HELP http_server_requests_seconds
# TYPE http_server_requests_seconds summary
http_server_requests_seconds_count{exception="None",method="GET",outcome="SUCCESS",status="200",uri="/actua
http_server_requests_seconds_sum{exception="None",method="GET",outcome="SUCCESS",status="200",uri="/actuato
# HELP http_server_requests_seconds_max
# TYPE http_server_requests_seconds_max gauge
http_server_requests_seconds_max{exception="None",method="GET",outcome="SUCCESS",status="200",uri="/actuato
# HELP jvm_buffer_count_buffers An estimate of the number of buffers in the pool
# TYPE jvm_buffer_count_buffers gauge
jvm_buffer_count_buffers{id="mapped - 'non-volatile memory'",} 0.0
jvm_buffer_count_buffers{id="mapped",} 0.0
jvm_buffer_count_buffers{id="direct",} 5.0
# HELP process_files_max_files The maximum file descriptor count
# TYPE process_files_max_files gauge
```

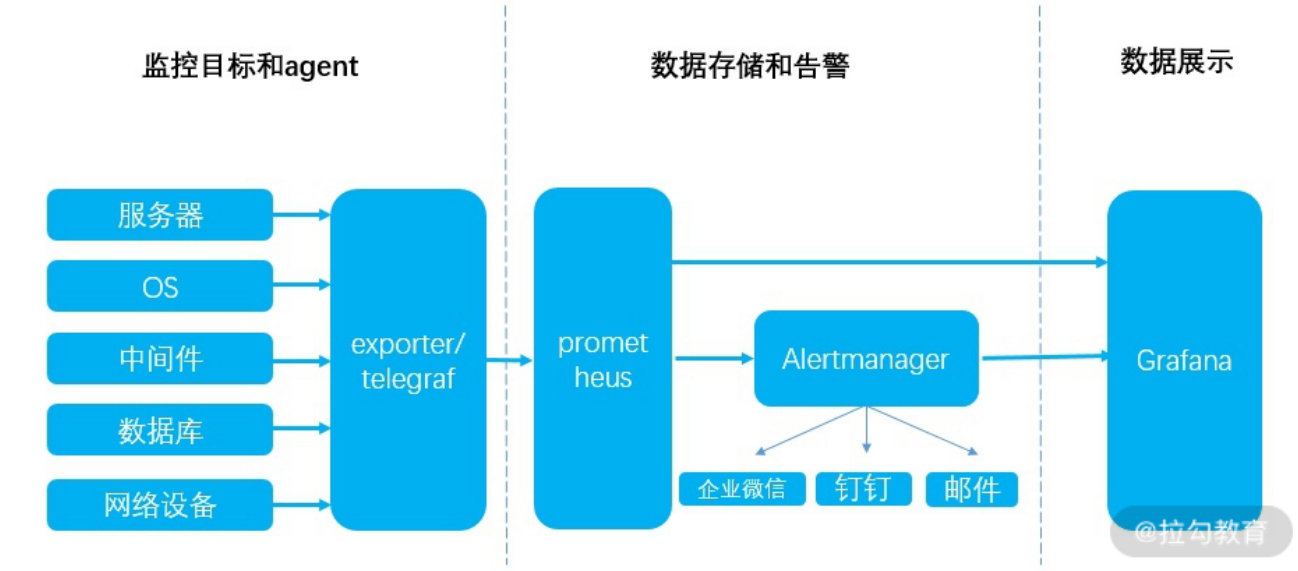

想要监控业务数据也是比较简单的，你只需要注入一个 MeterRegistry 实例即可，下面是一段示例代码：

```
@Autowired
MeterRegistry registry;

@GetMapping("/test")
@ResponseBody
public String test() {
    registry.counter("test",
            "from", "127.0.0.1",
            "method", "test"
    ).increment();

    return "ok";
}
```

从监控连接中，我们可以找到刚刚添加的监控信息。

```
test_total{from="127.0.0.1",method="test",} 5.0
```

这里简单介绍一下流行的**Prometheus 监控体系**，Prometheus 使用拉的方式获取监控数据，这个暴露数据的过程可以交给功能更加齐全的 telegraf 组件。



如上图，我们通常使用 Grafana 进行监控数据的展示，使用 AlertManager 组件进行提前预警。这一部分的搭建工作不是我们的重点，感兴趣的同学可自行研究。

下图便是一张典型的监控图，可以看到 Redis 的缓存命中率等情况。

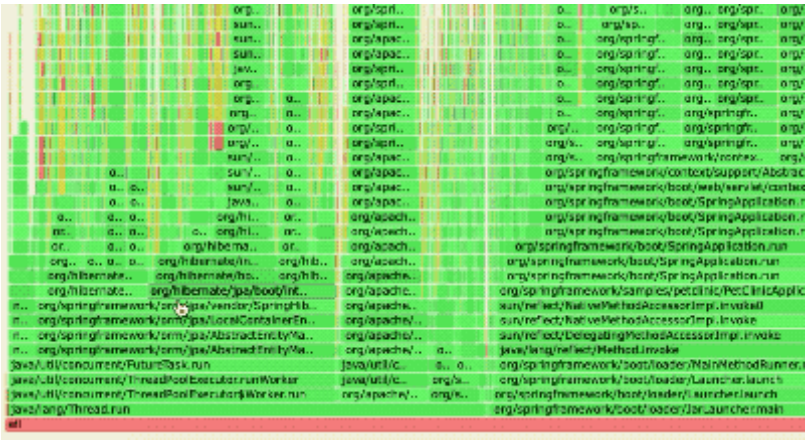
Java 生成火焰图

火焰图是用来分析程序运行瓶颈的工具。

火焰图也可以用来分析 Java 应用。可以从 github 上下载 async-profiler 的压缩包进行相关操作。比如，我们把它解压到 /root/ 目录，然后以 javaagent 的方式来启动 Java 应用，命令行如下：

```
java -agentpath:/root/build/libasyncProfiler.so=start,svg,file=profile.svg -j
```

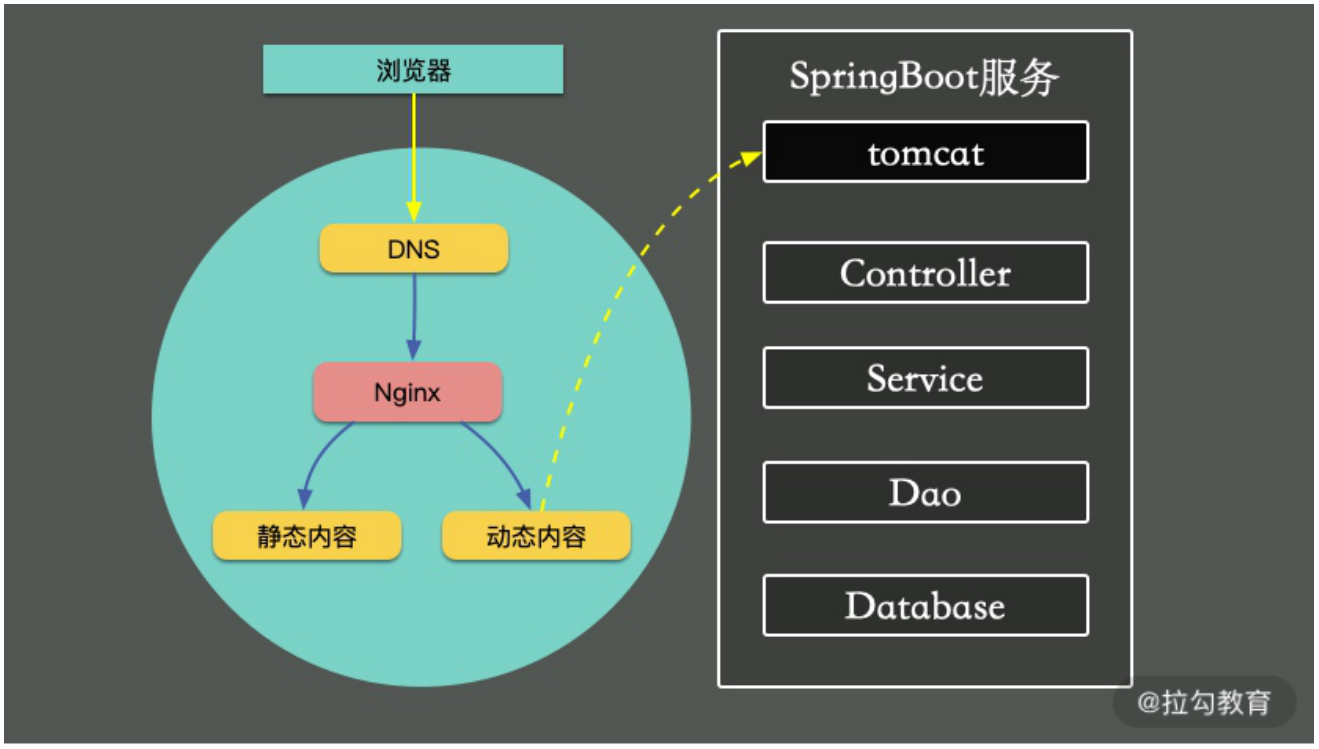
运行一段时间后，停止进程，可以看到在当前目录下，生成了 profile.svg 文件，这个文件是可以用浏览器打开的。 如下图所示，纵向，表示的是调用栈的深度；横向，表明的是消耗的时间。所以格子的宽度越大，越说明它可能是一个瓶颈。一层层向下浏览，即可找到需要优化的目标。



优化思路

对一个普通的 Web 服务来说，我们来看一下，要访问到具体的数据，都要经历哪些主要的环节？

如下图，在浏览器中输入相应的域名，需要通过 DNS 解析到具体的 IP 地址上，为了保证高可用，我们的服务一般都会部署多份，然后使用 Nginx 做反向代理和负载均衡。



Nginx 根据资源的特性，会承担一部分动静分离的功能。其中，动态功能部分，会进入我们的SpringBoot 服务。

SpringBoot 默认使用内嵌的 tomcat 作为 Web 容器，使用典型的 MVC 模式，最终访问到我们的数据。

HTTP 优化

下面我们举例来看一下，哪些动作能够加快网页的获取。为了描述方便，我们仅讨论 HTTP1.1 协议的。

1.使用 CDN 加速文件获取

比较大的文件，尽量使用 CDN（Content Delivery Network）分发，甚至是一些常用的前端脚本、样式、图片等，都可以放到 CDN 上。CDN 通常能够加快这些文件的获取，网页加载也更加迅速。

2.合理设置 Cache-Control 值

浏览器会判断 HTTP 头 Cache-Control 的内容，用来决定是否使用浏览器缓存，这在管理一些静态文件的时候，非常有用，相同作用的头信息还有 Expires。Cache-Control 表示多久之后过期；Expires 则表示什么时候过期。

这个参数可以在 Nginx 的配置文件中进行设置。

```
location ~* ^.+\.(ico|gif|jpg|jpeg|png)$ {
            # 缓存1年
            add_header Cache-Control: no-cache, max-age=31536000;
}
```

3.减少单页面请求域名的数量

减少每个页面请求的域名数量，尽量保证在 4 个之内。这是因为，浏览器每次访问后端的资源，都需要先查询一次 DNS，然后找到 DNS 对应的 IP 地址，再进行真正的调用。

DNS 有多层缓存，比如浏览器会缓存一份、本地主机会缓存、ISP 服务商缓存等。从 DNS 到 IP 地址的转变，通常会花费 20-120ms 的时间。减少域名的数量，可加快资源的获取。

4.开启 gzip

开启 gzip，可以先把内容压缩后，浏览器再进行解压。由于减少了传输的大小，会减少带宽的使用，提高传输效率。

在 nginx 中可以很容易地开启，配置如下：

```
gzip on;
gzip_min_length 1k;
```

```
gzip_buffers 4 16k;
gzip_comp_level 6;
gzip_http_version 1.1;
gzip_types text/plain application/javascript text/css;
```

5.对资源进行压缩

对 JavaScript 和 CSS，甚至是 HTML 进行压缩。道理类似，现在流行的前后端分离模式，一般都是对这些资源进行压缩的。

6.使用 keepalive

由于连接的创建和关闭，都需要耗费资源。用户访问我们的服务后，后续也会有更多的互动，所以保持长连接可以显著减少网络交互，提高性能。

nginx 默认开启了对客户端的 keep avlide 支持，你可以通过下面两个参数来调整它的行为。

```
http {
    keepalive_timeout  120s 120s;
    keepalive_requests 10000;
}
```

nginx 与后端 upstream 的长连接，需要手工开启，参考配置如下：

```
location ~ /{
       proxy_pass http://backend;
       proxy_http_version 1.1;
       proxy_set_header Connection "";
}
```

自定义 Web 容器

如果你的项目并发量比较高，想要修改最大线程数、最大连接数等配置信息，可以通过自定义Web 容器的方式，代码如下所示。

```
@SpringBootApplication(proxyBeanMethods = false)
public class App implements WebServerFactoryCustomizer<ConfigurableServletWeb
    public static void main(String[] args) {
        SpringApplication.run(PetClinicApplication.class, args);
    }
    @Override
    public void customize(ConfigurableServletWebServerFactory factory) {
        TomcatServletWebServerFactory f = (TomcatServletWebServerFactory) fac
        f.setProtocol("org.apache.coyote.http11.Http11Nio2Protocol");

        f.addConnectorCustomizers(c -> {
            Http11NioProtocol protocol = (Http11NioProtocol) c.getProtocolHan
            protocol.setMaxConnections(200);
            protocol.setMaxThreads(200);
            protocol.setSelectorTimeout(3000);
```

```
            protocol.setSessionTimeout(3000);
            protocol.setConnectionTimeout(3000);
        });
    }
}
```

注意上面的代码，我们设置了它的协议为 org.apache.coyote.http11.Http11Nio2Protocol，意思就是开启了 Nio2。这个参数在 Tomcat 8.0之后才有，开启之后会增加一部分性能。 对比如下（测试项目代码见 spring-petclinic-main）：

默认。

```
[root@localhost wrk2-master]# ./wrk -t2 -c100 -d30s -R2000 http://172.16.1.57
Running 30s test @ http://172.16.1.57:8080/owners?lastName=
  2 threads and 100 connections
  Thread calibration: mean lat.: 4588.131ms, rate sampling interval: 16277ms
  Thread calibration: mean lat.: 4647.927ms, rate sampling interval: 16285ms
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    16.49s     4.98s   27.34s    63.90%
    Req/Sec   106.50      1.50   108.00    100.00%
  6471 requests in 30.03s, 39.31MB read
  Socket errors: connect 0, read 0, write 0, timeout 60
Requests/sec:    215.51
Transfer/sec:      1.31MB
```

Nio2。

```
[root@localhost wrk2-master]# ./wrk -t2 -c100 -d30s -R2000 http://172.16.1.57
Running 30s test @ http://172.16.1.57:8080/owners?lastName=
  2 threads and 100 connections
  Thread calibration: mean lat.: 4358.805ms, rate sampling interval: 15835ms
  Thread calibration: mean lat.: 4622.087ms, rate sampling interval: 16293ms
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    17.47s     4.98s   26.90s    57.69%
    Req/Sec   125.50      2.50   128.00    100.00%
  7469 requests in 30.04s, 45.38MB read
  Socket errors: connect 0, read 0, write 0, timeout 4
Requests/sec:    248.64
Transfer/sec:      1.51MB
```

你甚至可以将 tomcat 替换成 undertow。undertow 也是一个 Web 容器，更加轻量级一些，占用的内存更少，启动的守护进程也更少，更改方式如下：

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
      <exclusions>
        <exclusion>
          <groupId>org.springframework.boot</groupId>
          <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
```

```
      </exclusions>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-undertow</artifactId>
    </dependency>
```

其实，对于 tomcat 优化最为有效的，还是 JVM 参数的配置，你可以参考上一课时的内容进行调整。 比如，使用下面的参数启动，QPS 由 248 上升到 308。

```
-XX:+UseG1GC -Xmx2048m -Xms2048m -XX:+AlwaysPreTouch
```

Skywalking

对于一个 web 服务来说，最缓慢的地方就在于数据库操作。所以，使用“07 | 案例分析：无处不在的缓存，高并发系统的法宝”和“08 | 案例分析：Redis 如何助力秒杀业务”提供的本地缓存和分布式缓存优化，能够获得最大的性能提升。

对于如何定位到复杂分布式环境中的问题，我这里想要分享另外一个工具：Skywalking。

Skywalking 是使用探针技术（JavaAgent）来实现的。通过在 Java 的启动参数中，加入 javaagent 的 Jar 包，即可将性能数据和调用链数据封装，并发送到 Skywalking 的服务器。

下载相应的安装包（如果使用 ES 存储，需要下载专用的安装包），配置好存储之后，即可一键启动。

将 agent 的压缩包，解压到相应的目录。

```
tar xvf skywalking-agent.tar.gz  -C /opt/
```

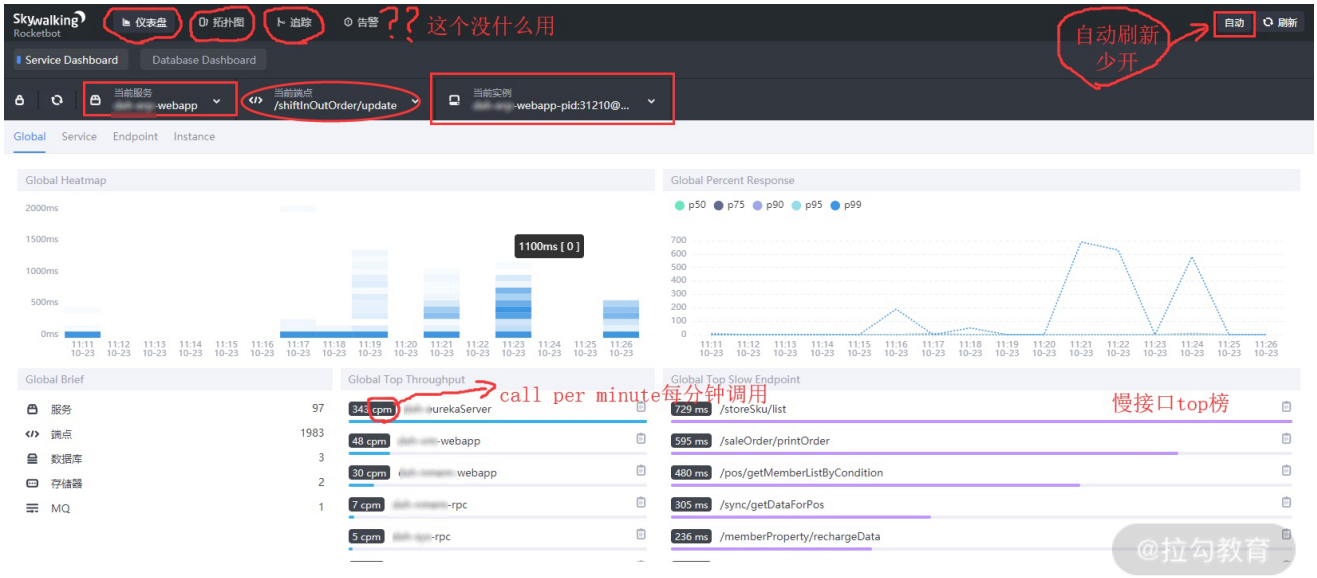
在业务启动参数中加入 agent 的包。比如，原来的启动命令是：

```
java  -jar /opt/test-service/spring-boot-demo.jar  --spring.profiles.active=d
```

改造后的启动命令是：

```
java -javaagent:/opt/skywalking-agent/skywalking-agent.jar -Dskywalking.agent
```

访问一些服务的链接，打开 Skywalking 的 UI，即可看到下图的界面。这些指标可以类比“01 | 理论分析：性能优化，有哪些衡量指标？需要注意什么？”提到的衡量指标去理解，我们就可以从图中找到响应比较慢 QPS 又比较高的接口，进行专项优化。



各个层次的优化方向

1.Controller 层

controller 层用于接收前端的查询参数，然后构造查询结果。现在很多项目都采用前后端分离的架构，所以 controller 层的方法，一般会使用 @ResponseBody 注解，把查询的结果，解析成 JSON 数据返回（兼顾效率和可读性）。

由于 controller 只是充当了一个类似功能组合和路由的角色，所以这部分对性能的影响就主要体现在数据集的大小上。如果结果集合非常大，JSON 解析组件就要花费较多的时间进行解析，

大结果集不仅会影响解析时间，还会造成内存浪费。

假如结果集在解析成 JSON 之前，占用的内存是 10MB，那么在解析过程中，有可能会使用 20M 或者更多的内存去做这个工作。

我见过很多案例，由于返回对象的嵌套层次太深、引用了不该引用的对象（比如非常大的 byte[] 对象），造成了内存使用的飙升。

所以，**对于一般的服务，保持结果集的精简，是非常有必要的**，这也是 DTO（data transfer object）存在的必要。如果你的项目，返回的结果结构比较复杂，对结果集进行一次转换是非常有必要的。

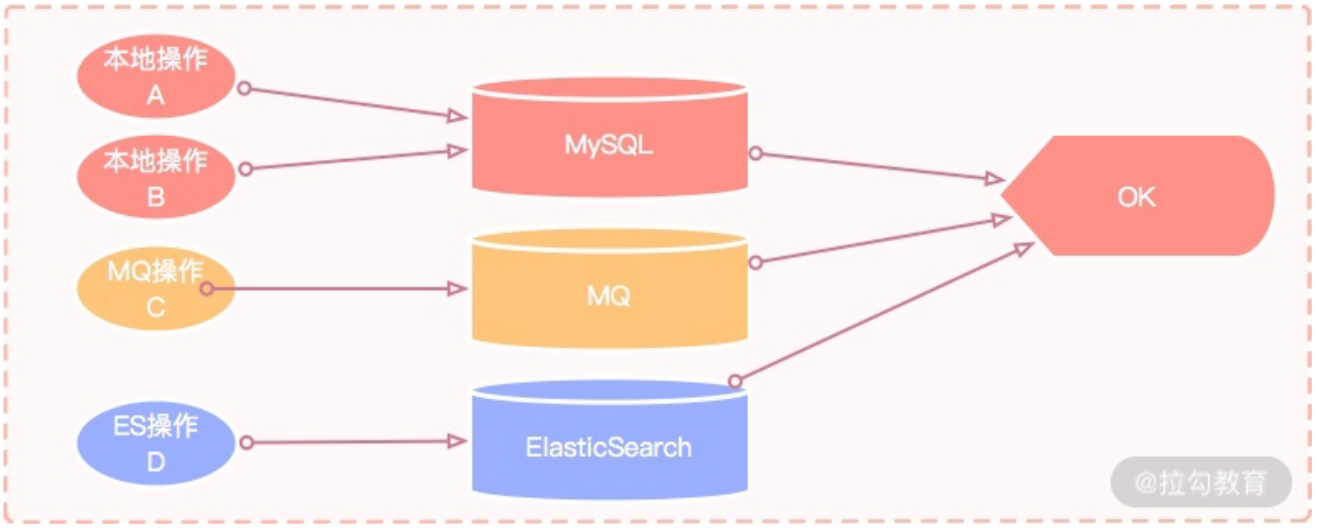
2.Service 层

service 层用于处理具体的业务，大部分功能需求都是在这里完成的。service 层一般是使用单例模式（prototype），很少会保存状态，而且可以被 controller 复用。

service 层的代码组织，对代码的可读性、性能影响都比较大。我们常说的设计模式，大多数都是针对 service 层来说的。

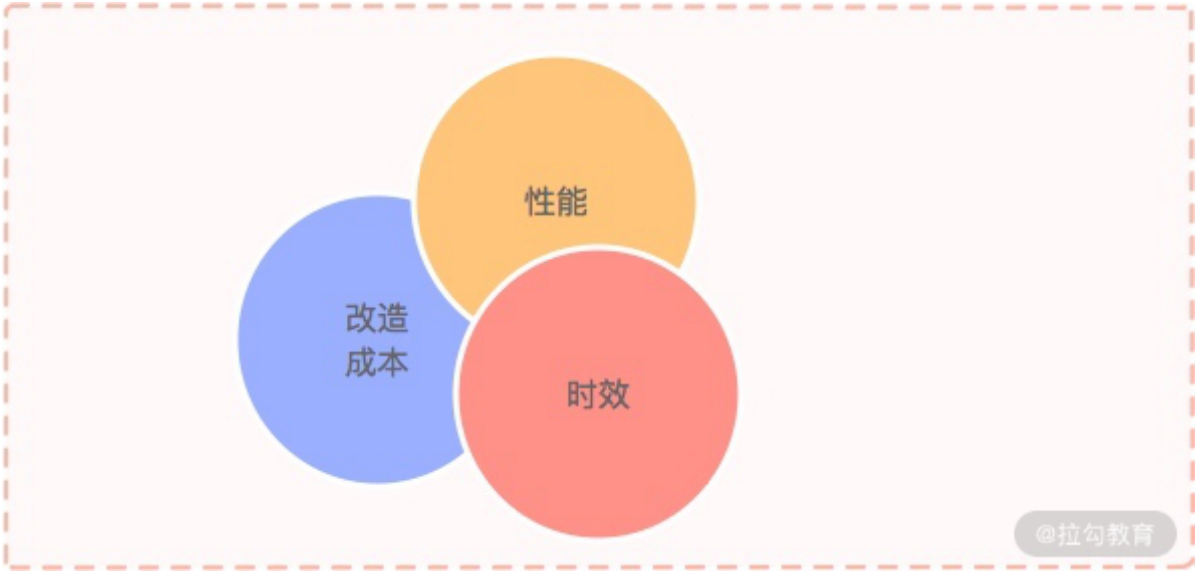
service 层会频繁使用更底层的资源，通过组合的方式获取我们所需要的数据，大多数可以通过我们前面课时提供的优化思路进行优化。

这里要着重提到的一点，就是分布式事务。



如上图，四个操作分散在三个不同的资源中。要想达到一致性，需要三个不同的资源 MySQL、MQ、ElasticSearch 进行统一协调。它们底层的协议，以及实现方式，都是不一样的，那就无法通过 Spring 提供的 Transaction 注解来解决，需要借助外部的组件来完成。

很多人都体验过，加入了一些保证一致性的代码，一压测，性能掉的惊掉下巴。分布式事务是性能杀手，因为它要使用额外的步骤去保证一致性，常用的方法有：两阶段提交方案、TCC、本地消息表、MQ 事务消息、分布式事务中间件等。



如上图，分布式事务要在改造成本、性能、时效等方面进行综合考虑。有一个介于分布式事务和非事务之间的名词，叫作**柔性事务**。柔性事务的理念是将业务逻辑和互斥操作，从资源

层上移至业务层面。

关于传统事务和柔性事务，我们来简单比较一下。

ACID

关系数据库, 最大的特点就是事务处理, 即满足 ACID。

- 原子性（Atomicity）：事务中的操作要么都做，要么都不做。
- 一致性（Consistency）：系统必须始终处在强一致状态下。
- 隔离性（Isolation）：一个事务的执行不能被其他事务所干扰。
- 持久性（Durability）：一个已提交的事务对数据库中数据的改变是永久性的。

BASE

BASE 方法通过牺牲一致性和孤立性来提高可用性和系统性能。

BASE 为 Basically Available、Soft-state、Eventually consistent 三者的缩写，其中 BASE 分别代表：

- 基本可用（Basically Available）：系统能够基本运行、一直提供服务。
- 软状态（Soft-state）：系统不要求一直保持强一致状态。
- 最终一致性（Eventual consistency）：系统需要在某一时刻后达到一致性要求。

互联网业务，推荐使用补偿事务，完成最终一致性。比如，通过一系列的定时任务，完成对数据的修复。

3.Dao 层

经过合理的数据缓存，我们都会尽量避免请求穿透到 Dao 层。除非你对 ORM 本身提供的缓存特性特别的熟悉；否则，都推荐你使用更加通用的方式去缓存数据。

Dao 层，主要在于对 ORM 框架的使用上。比如，在 JPA 中，如果加了一对多或者多对多的映射关系，而又没有开启懒加载，级联查询的时候就容易造成深层次的检索，造成了内存开销大、执行缓慢的后果。

在一些数据量比较大的业务中，多采用分库分表的方式。在这些分库分表组件中，很多简单的查询语句，都会被重新解析后分散到各个节点进行运算，最后进行结果合并。

举个例子，select count(*) from a 这句简单的 count 语句，就可能将请求路由到十几张表中去运算，最后在协调节点进行统计，执行效率是可想而知的。目前，分库分表中间件，比较

有代表性的是驱动层的 ShardingJdbc 和代理层的 MyCat，它们都有这样的问题。这些组件提供给使用者的视图是一致的，但我们在编码的时候，一定要注意这些区别。

小结

下面我们来总结一下。

本课时，我们简单看了一下 SpringBoot 常见的优化思路，然后介绍了三个新的性能分析工具。

- 一个是监控系统 Prometheus，可以看到一些具体的指标大小；
- 一个是火焰图，可以看到具体的代码热点；
- 一个是 Skywalking，可以分析分布式环境中的调用链。

SpringBoot 自身的 Web 容器是 Tomcat，那我们就可以通过对 Tomcat 的调优来获取性能提升。当然，对于服务上层的负载均衡 Nginx，我们也提供了一系列的优化思路。

最后，我们看了在经典的 MVC 架构下，Controller、Service、Dao 的一些优化方向，并着重看了 Service 层的分布式事务问题。

SpringBoot 作为一个广泛应用的服务框架，在性能优化方面已经做了很多工作，选用了很多高速组件。比如，数据库连接池默认使用 hikaricp，Redis 缓存框架默认使用 lettuce，本地缓存提供 caffeine 等。对于一个普通的数据库交互的 Web 服务来说，缓存是最主要的优化手段。

但细节决定成败，05-19 课时的内容对性能优化都有借鉴意义。下一课时（也就是咱们专栏的最后一课时），我将从问题发现、目标制定、优化方式上进行整体性的总结。

上一页 下一页