7-嵌入式容器的配置与应用

1.嵌入式容器的运行参数配置

在Spring Boot项目中，可以支持Tomcat、Jetty、Undertow的Web应用服务容器。

当我们添加了spring-boot-starter-web依赖后，默认会使用Tomcat作为嵌入式Web容器，不需要我们单独部署，将Web应用打成jar包即可运行。

调整SpringBoot应用容器的参数两种配置方法

- 修改配置文件（简单）
- 自定义配置类 （专业调优）

一、配置文件方式

在application.properties / application.yml可以配置Web 容器运行所需要的属性，可以通过该链接在官方网站查看关于server的所有配置项：server-properties。

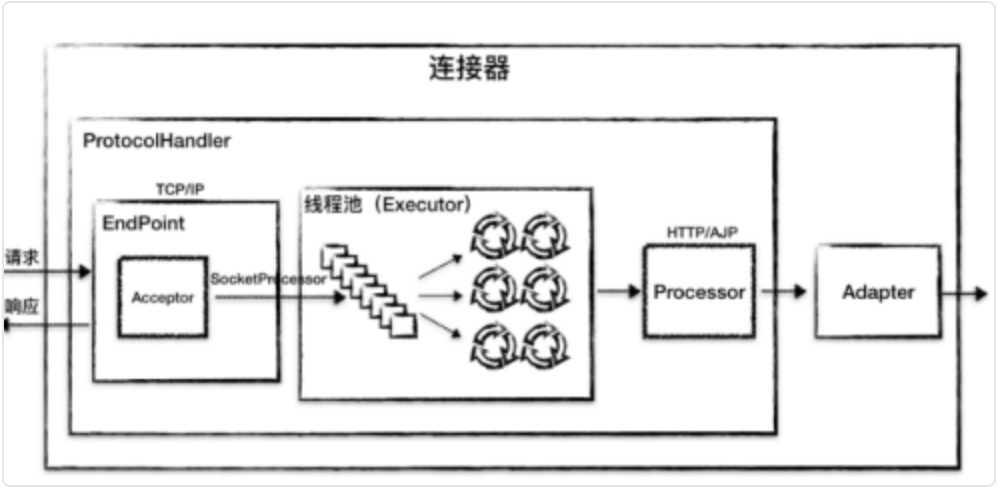
- server.xx开头的是所有servlet容器通用的配置，
- server.tomcat.xx开头的是tomcat 容器特有的配置参数参数
- server.jetty.xx开头的是Jetty 容器特有的配置参数参数
- server.undertow.xx开头的是undertow容器特有的配置参数参数

1.1.常用配置参数

参数	默认值	说明
server.port	8080	配置Web容器的端口号
server.servlet.session.timeout	30m(30分钟)	session失效时间。如果不写单位则默认单位是秒。（注意：由于Tomcat中配置session过期时间是以分钟为单位，如果这里设置是秒的话，那么会自动转换为一个不超过所配置秒数的最大分钟数。比如配置了119秒(1分59秒)，那么实际session过期时间是1分钟）
server.servlet.context-path	/	URL访问路径的基础路径
server.tomcat.uri-encoding	UTF-8	配置Tomcat请求编码
server.tomcat.basedir		配置Tomcat运行日志和临时文件的目录。若不配置，则默认使用系统的临时目录。

1.2. tomcat性能优化核心参数

tomcat连接器工作原理图：



- 在Acceptor之前维护一个请求接收队列，该队列的最大长度即：tomcat可以接受的最大请求连接数：server.tomcat.max-connections。
- Acceptor监听连接请求，并生成一个 SocketProcessor 任务提交到线程池去处理

- 当线程池里面的所有线程都被占用，新建的SocketProcessor 任务被放入等待队列，即：server.tomcat.accept-count
- 线程池的server.tomcat.threads.max决定了tomcat的极限SocketProcessor 任务处理能力。不是越大越好，线程越多耗费的资源也越多。
- 线程池的server.tomcat.threads.min-spare在应用空闲时，保留一定的线程数在线程池内。避免请求到来后，临时创建线程浪费时间。

参数	默认值	说明
server.tomcat.max-connections	8192	接受的最大请求连接数
server.tomcat.accept-count	100	当所有的线程都被占用，被放入请求队列等待的最大的请求连接数量
server.tomcat.threads.max	200	最大的工作线程池数量
server.tomcat.threads.min-spare	10	最小的工作线程池数量

二、自定义配置类方式

步骤：

1.建立一个配置类，加上@Configuration注解

2.添加定制器ConfigurableServletWebServerFactory

3.将定制器返回

```java
@Configuration
public class TomcatCustomizer {

    @Bean
    public ConfigurableServletWebServerFactory configurableServletWebServerFactory(){
        TomcatServletWebServerFactory factory = new TomcatServletWebServerFactory();
        factory.addConnectorCustomizers(new MyTomcatConnectionCustomizer());
        return factory;
    }


    static class MyTomcatConnectionCustomizer implements TomcatConnectorCustomizer {

        public MyTomcatConnectionCustomizer() {
        }

        @Override
        public void customize(Connector connector) {
            connector.setPort(Integer.parseInt("8888"));
            connector.setProperty("maxConnections", "8192");
            connector.setProperty("acceptorThreadCount", "100");
            connector.setProperty("minSpareThreads", "10");
            connector.setProperty("maxThreads", "200");
        }
    }
}
```

这种方法可定制的内容更多，也更灵活。但需要深入理解server 容器的底层实现原理及设计机制，也需要具备一定的TomcatServletWebServerFactory的API熟练度。

2.为Web容器配置HTTPS

HTTPS是HTTP协议的安全版本，旨在提供数据传输层安全性（TLS）。当你的应用不使用HTTPS的时候，浏览器地址栏就会出现一个不安全的提示。HTTPS加密每个数据包以安全方式进行传输，并保护敏感数据免受窃听者或黑客的攻击。

可以通过在Web应用程序上安装SSL证书来实现HTTPS，互联网上受信任的证书通常是需要（CA）认证机构颁发的证书（通常是收费的）。一个标准的SSL证书，还是有点小贵的。国内的一些厂商虽然可以提供免费的证书，但是都有一定的免费时效性限制。

如果是以学习为目的，我们也可以使用自签名证书，即：使用Java Keytool生成自签名证书。完全不需要购买CA机构认证的SSL证书。

一、如何生成自签名证书

管理员身份启动命令行，使用如下的keytool命令生成自签名证书：

keytool –genkeypair –alias selfsigned_localhost_sslserver –keyalg RSA –keysize 2048 –storetype PKCS12 –keystore zimug-ssl-key.p12 –validity 3650

自签名证书受密码保护。命令回车之后，会提示输入密码（这个密码要记住，后面会用到）和其他详细信息。

完成上述步骤后，便会创建PKS密钥并将其存储在当前目录下。

> 命令参数说明：
>
> - –genkey：表示要创建一个新的密钥

- -alias：表示keystore的别名
- -keyalg：表示使用的加密算法是RSA（一种非对称加密算法）
- -keysize：表示密钥的长度
- -keystore：表示生成的密钥存放位置
- -validity：表示密钥的有效时间（单位为天）

三、将SSL应用于Spring Boot应用程序

1. 复制mqxu-ssl-key，将其放在应用根目录下。
2. 将SSL密钥信息添加到application.yml中。

YAML | 复制代码

```yaml
server:
  port: 8888
  ssl:
    key-store: mqxu-ssl-key.p12
    key-store-password: 123456
    key-store-type: PKCS12
```

四、测试

此时如果我们继续使用http协议去访问应用资源，会得到如下的响应信息：

Bash | 复制代码

```
Bad Request
This combination of host and port requires TLS.
```

使用HTTPS协议去访问应用资源，https://localhost:8888/hello。才会得到正确的结果。

五、将HTTP请求重定向为HTTPS

首先配置两个服务端口，server.port是我们真正的服务端口，即HTTPS服务端口。另外再定义一个server.httpPort,当客户端访问该HTTP协议端口的时候，自动跳转到HTTPS服务端口。

```Bash
server:
  port: 8888
  httpPort: 80
```

需要使用到上一节使用编码方式进行配置的方法。下面的配置类不用改。

Java | 复制代码

```java
@Configuration
public class TomcatCustomizer {

    @Value("${server.httpPort}")
    int httpPort;
    @Value("${server.port}")
    int httpsPort;


    @Bean
    public ConfigurableServletWebServerFactory configurableServletWebServerFactory(){
        TomcatServletWebServerFactory factory = new TomcatServletWebServerFactory(){
            @Override
            protected void postProcessContext(Context context) {
                SecurityConstraint constraint = new SecurityConstraint();
                constraint.setUserConstraint("CONFIDENTIAL");

                SecurityCollection collection = new SecurityCollection();
                collection.addPattern("/*");
                constraint.addCollection(collection);
                context.addConstraint(constraint);
            }
        };;
        factory.addAdditionalTomcatConnectors(connector());
        //这里填充配置
        return factory;
    }


    public Connector connector() {
        Connector connector = new Connector("org.apache.coyote.http11.Http11NioProtocol");
        connector.setScheme("http");
        //Connector监听的http的端口号
        connector.setPort(httpPort);
        connector.setSecure(false);
        //监听到http的端口号后转向到的https的端口号
        connector.setRedirectPort(httpsPort);
        return connector;
    }

}
```

这样当我们通过HTTP协议：http://localhost:80/hello 的时候，浏览器访问地址就会自动的跳转到HTTPS连接器服务端口 https://localhost:8888/hello

3.切换到jetty&undertow容器

本节介绍将SpringBoot默认的Tomcat容器切换为jetty或者undertow。虽然可以使用jetty或者undertow替换掉tomcat，但是不建议这么做，但是jetty与undertow的NIO模型还是有必要学一下的，这也是绝大部分Web应用中间件提供网络服务的IO模型。

可能在某些场景下，jetty或者undertow的测试结果的某些指标会好于tomcat。但是tomcat 综合各方面条件来说，无论从性能、稳定性、资源利用率来说都是比较优秀的。

一、替换掉tomcat

SpringBoot默认是使用tomcat作为默认的应用容器。如果需要把tomcat替换为jetty或者undertow，需要先把tomcat相关的jar包排除出去。如下代码所示

XML | 复制代码

```xml
<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>
```

如果使用Jetty容器，那么添加

XML | 复制代码

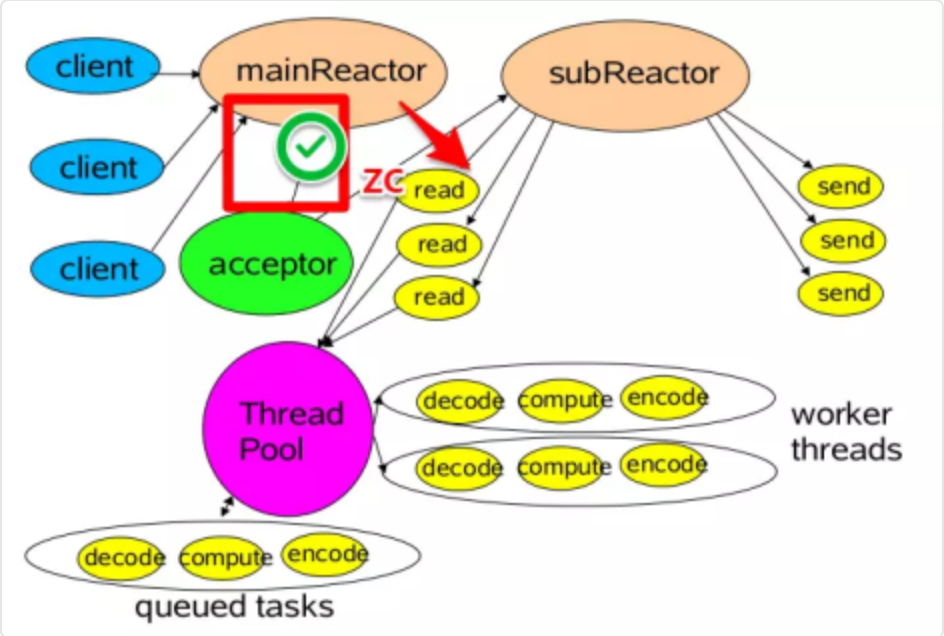
```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

如果使用Undertow容器，那么添加

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

如果不做特殊的调优配置，全部使用默认值的话，我们的替换工作就已经完成了。

二、Reactor NIO多线程模型



1. mainReactor负责监听server socket，用来处理新连接的建立，将建立的socketChannel指定注册给subReactor。
2. subReactor维护自己的selector, 基于mainReactor 注册的socketChannel多路分离IO读写事件，读写网络数据，对业务处理的功能，将其扔给worker线程池来完成实际的请求任务处理。

三、切换为 Jetty Server

常用jetty调优配置参数

参数	默认值	说明
server.jetty.threads.acceptors	-1.0	acceptor线程的数量，acceptor是用于连接接收的连接器。当设置成-1的时候，会根据CPU的逻辑核数/8来决定，最大不能超过4个
server.jetty.threads.selectors	-1.0	selector线程的数量. 当设置成-1的时候，根据CPU的逻辑核数/2，最少1个。
server.jetty.threads.min	8	worker工作线程池最小线程数量.
server.jetty.threads.max	200	worker工作线程池最大线程数量

三、切换到undertow

下文配置中的io-threads可以认为是acceptor线程数，用来处理连接的建立。

worker-threads就是工作线程池的线程数量，用来处理实际请求任务。

YAML | 复制代码

```yaml
server:
  port: 8888
  # 下面是配置undertow作为服务器的参数
  undertow:
    # 设置IO线程数, 它主要执行非阻塞的任务,它们会负责多个连接, 默认设置每个CPU核心一个线程
    io-threads: 4
    # 工作任务线程池, 默认为io-threads的8倍
    worker-threads: 32
```

4.打war包部署到外置tomcat容器

一、修改打包方式

XML | 复制代码

```xml
<packaging>war</packaging>
```

二、 排除内置tomcat的依赖

使用外置的tomcat，要将内置的嵌入式tomcat的相关jar排除。

XML 复制代码

```xml
<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>
```

三、新增加一个类继承SpringBootServletInitializer实现configure：

SpringBootServletInitializer源码注释：

> Note that a WebApplicationInitializer is only needed if you are building a war file and deploying it. If you prefer to run an embedded web server then you won't need this at all.

如果你正在构建WAR文件并部署，则需要WebApplicationInitializer。如果你喜欢运行一个嵌入式Web服务器，那么不需要这个。

Java 复制代码

```java
public class ServletInitializer extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder builder) {
        //此处的Application.class为带有@SpringBootApplication注解的启动类
        return builder.sources(BootLaunchApplication.class);
    }
}
```

注意事项：

使用外部Tomcat部署访问的时候，application.properties(或者application.yml)中的如下配置将失效，请使用外置的tomcat的端口，tomcat的webapps下项目名进行访问。

```yaml
server.port= server.servlet.context-path=
```

四、build要有finalName标签

pom.xml中的构建build代码段，要有应用最终构建打包的名称。

```xml
    <finalName>boot-launch</finalName>
```

五、打包与运行

war方式打包，打包结果将存储在项目的target目录下面。

```bash
mvn clean package -Dmaven.test.skip=true
```

然后将war包copy到外置Tomcat webapps目录里。

在外置tomcat中运行：${Tomcat_home}/bin/目录下执行startup.bat(windows)或者startup.sh(linux)，然后通过浏览器访问应用，测试效果。

需要注意的是

- 在boot-launch.war在tomcat webapps目录里面解压到boot-launch文件夹。所以访问应用的时候，必须使用http://localhost:8888/boot-launch/template/thymeleaf
  不能是：http://localhost:8888/template/thymeleaf，会报404错误。
- 静态资源引用也必须是：/boot-launch/image/xxxx.png，不能是/image/xxxx.png
- JSP的war包中，webjars的资源使用方式不再被支持