Spring Session

Rob Winch, Vedran Pavić, Jakub Kubrynski

Version 1.3.1.RELEASE

关于本翻译

这是本人工作之余翻译的，方便自己使用，也贡献出来方便其他人。有翻译不正确的地方请指正。如果您觉得本文对您有所帮助，不妨打赏小弟一杯咖啡。

Spring Session提供了一个用于管理用户会话信息的API和实现。

1.简介

Spring Session提供了一个用于管理用户会话信息的API和实现。 它还提供了与HttpSession和WebSocket透明的集成：

nHttpSession -允许在应用程序容器（即Tomcat）中替换HttpSession。 其他功能包括：

l集群Session支持 - Spring Session使得集群Session非常简单，这使得集群Session的解决方案不依赖于应用程序容器特定的解决方案。

l多个浏览器会话支持 - Spring Session支持在单个浏览器实例中管理多个用户的会话（即与Google类似的多个经过身份验证的帐户）。

lRESTful API支持 - Spring Session允许在Http请求头中提供会话ID以使用RESTful API

nWebSocket -提供了在接收WebSocket消息时保持HttpSession处于激活状态的功能。

2.Spring Session1.3的新特性

以下是Spring Session 1.3中主要的新功能。 如果需要了解完整的更新情况，请参考1.3.0.M1，1.3.0.M2，1.3.0.RC1和1.3.0.RELEASE的更新日志。

- 支持Hazelcast
- 支持Spring Security的并发会话管理
- 添加了OrientDB社区扩展。
- GenericJackson2JsonRedisSerializer参考示例和Spring Security的新Jackson支持
- Lettuce参考指南
- spring.session.cleanup.cron.expression可用于覆盖清理任务的cron表达式。
- 许多性能上的改进和bug修复

3.示例与入门指南

如果您初次使用Spring Session，那么最好参考我们提供的示例程序。

来源	描述	参考
HttpSession	演示Spring Session来替换HttpSession，使用Redis作为Session存储容器。	HttpSession Guide
HttpSession XML	演示Spring Session基于XML的配置方式来替HttpSession， 使用Redis作为Session存储容器。	HttpSession XML Guide
HttpSession with GemFire using Spring Boot	演示Spring Session在Spring Boot应用程序中使用Client / Server拓扑替换HttpSession，使用GemFire作为Session 存储容器。	
HttpSession with GemFire (Client/Server)	演示Spring Session使用Client / Server拓扑替HttpSession，使用GemFire作为Session 存储容器。	HttpSession GemFire Client/Server Guide
HttpSession with GemFire (Client/Server) using XML	演示Spring Session基于XML的配置方式，使用Client / Server拓扑替HttpSession，将GemFire作为Session 存储容器。	HttpSession GemFire Client/Server XML Guide
HttpSession with GemFire (P2P)	演示Spring Session使用P2P拓扑替HttpSession，将GemFire作为Session 存储容器。	HttpSession GemFire P2P Guide
HttpSession with GemFire (P2P) using XML	演示Spring Session基于XML的配置方式，使用P2P拓扑替换HttpSession，将GemFire作为Session 存储容器。	HttpSession GemFire P2P XML Guide
自定义Cookie	演示Spring Session如何使用自定义Cookie	Custom Cookie Guide
Spring Boot	演示如何在Spring boot应用中使用Spring Session。	Spring Boot Guide
Grails 3	演示在Grails3中使用Spring session	Grails 3 Guide
Spring Security	演示如何在Spring Security应用中使用Spring session	Spring Security Guide
REST	演示如何在REST应用程序中使用Spring Session来支持使用Http请求头进行身份验证。	REST Guide
Find by Username	演示如何使用Spring Session通过用户名查找会话。	Find by Username
多用户会话管理	演示如何使用Spring Session同时管理多个浏览器会话（例如Google帐户）。	Manage Multiple Users Guide
WebSocket	演示在WebSockets中如何使用Spring Session与。	WebSocket Guide
Mongo	演示在Monogo程序中如何使用Spring Session。	Mongo Guide
Hazelcast	演示在Hazelcast程序中如何使用Spring Session。	TBD
Hazelcast Spring	演示在基于Spring Security的Hazelcast程序中如何使用Spring Session。	Hazelcast Spring Guide
HttpSession JDBC	演示如何使用Spring Session替换HttpSession,将关系数据库作为Session存储容器。	HttpSession JDBC Guide
HttpSession JDBC XML	演示Spring Session基于XML的配置方式替换Https session，将关系数据库作为Session存储容器	HttpSession JDBC XML Guide
HttpSession JDBC Spring Boot	演示在Spring boot应用中使用Spring Session替换 Https session，将关系数据库作为Session存储容器	HttpSession JDBC Spring Boot Guide

4.HttpSession的集成

Spring Session支持与HttpSession的透明集成。 这意味着开发人员可以使用Spring Session支持的实现来切换HttpSession实现。

4.1为什么讨论Spring Session&HttpSession?

我们已经提到Spring Session支持与HttpSession的透明集成，但是这对我们有什么好处呢?

- 集群Session支持 - Spring Session使得集群Session非常简单，这使得集群Session的解决方案不依赖于应用程序容器特定的解决方案。
- 多个浏览器会话支持 - Spring Session支持在单个浏览器实例中管理多个用户的会话（即与Google类似的多个经过身份验证的帐户）。
- RESTful API支持 - Spring Session允许在Http请求头中提供会话ID以使用RESTful API

4.2基于Redis的HttpSession实现

Spring Session通过添加Servlet过滤器来实现替换HttpSession的，可以通过两种方式配置过滤器：

- 基于java的配置方式
- 基于XML的配置方式

4.2.1基于java的配置方式

本节介绍如何使用Java的配置方式来实现基于Redis的HttpSession。

注意：HttpSession Sample提供了一个关于如何使用Java配置集成Spring Session和HttpSession的工作示例。 您可以阅读下面的集成基本步骤，但是当您与自己的应用程序集成时，建议您遵循详细的HttpSession指南。

Spring的java配置

添加所需的依赖关系后，我们可以创建我们的Spring配置。 Spring配置负责创建一个使用Spring Session支持的实现替换HttpSession实现的Servlet过滤器。 添加以下Spring配置：

```
@EnableRedisHttpSession ①
public class Config {

        @Bean
        public LettuceConnectionFactory connectionFactory() {
                return new LettuceConnectionFactory(); ②
        }
}
```

①@EnableRedisHttpSession注释创建一个名为springSessionRepositoryFilter的Spring Bean，该实例实现了Filter。 过滤器是负责替换由Spring Session支持的HttpSession实现的过程。 在这种情况下，Spring Session由Redis支持。

②我们创建一个将Spring Session连接到Redis Server的RedisConnectionFactory。 我们配置默认端口为（6379）。有关配置，请参考Spring Data Redis的参考文档。

Java Servlet容器的初始化

我们的Spring Configuration创建了一个名为springSessionRepositoryFilter的Spring Bean，改Bean实现了Filter接口。 springSessionRepositoryFilter bean负责使用Spring Session支持的自定义实现来替换HttpSession。

为了使我们的过滤器能够做到这一点，Spring需要加载我们的Config配置类。 最后，我们需要确保我们的Servlet容器（即Tomcat）为每个请求使用我们的springSessionRepositoryFilter。 幸运的是，Spring Session提供了一个名为AbstractHttpSessionApplicationInitializer的实用程序类，这两个步骤非常简单。 你可以在下面找到一个例子：

src/main/java/sample/Initializer.java

```
public class Initializer extends AbstractHttpSessionApplicationInitializer { ①

        public Initializer() {
                super(Config.class); ②
        }
}
```

> 我们的类（Initializer）的名称并不重要。 重要的是我们扩展AbstractHttpSessionApplicationInitializer。

①第一步是扩展AbstractHttpSessionApplicationInitializer。 这样可以确保springSessionRepositoryFilter针对每个请求都会在Servlet容器注册。

②AbstractHttpSessionApplicationInitializer还提供了一种机制，可以确保Spring加载我们的Config。

4.2.2基于XML的配置方式

本节介绍通过XML的配置方式来实现基于Redis的HttpSession。

> HttpSession XML Sample提供了一个关于如何使用XML配置集成Spring Session和HttpSession的示例。 您可以阅读下面的集成基本步骤，但是当与您自己的应用程序集成时，建议您遵循详细的HttpSession XML指南。

Spring的XML 配置

添加所需的依赖关系后，我们可以创建我们的Spring配置。 Spring配置负责创建一个使用Spring Session支持的实现替换HttpSession实现的Servlet过滤器。 添加以下Spring配置：

src/main/webapp/WEB-INF/spring/session.xml

```
①
<context:annotation-config/>
<bean class="org.springframework.session.data.redis.config.annotation.web.http.RedisHttpSessionConfiguration"/>

②
<bean class="org.springframework.data.redis.connection.lettuce.LettuceConnectionFactory"/>
```

①我们使用<context:annotation-config />和RedisHttpSessionConfiguration的组合，因为Spring Session还没有提供XML Namespace支持（参见gh-104）。 这将创建一个Spring Bean，名称为springSessionRepositoryFilter，该Bean实现Filter接口，是负责替换由Spring Session支持的HttpSession实现的过程。 在这种情况下，Spring Session由Redis支持。

②我们创建一个将Spring Session连接到Redis Server的RedisConnectionFactory。 我们配置默认端口为（6379）。有关配置，请参考Spring Data Redis的参考文档。

以XML方式配置Servlet容器的初始化

我们的Spring Configuration创建了一个名为springSessionRepositoryFilter的Spring Bean，改Bean实现了Filter接口。 springSessionRepositoryFilter bean负责使用Spring Session支持的自定义实现来替换HttpSession。

为了使我们的过滤器能够实现这种功能，我们需要指示Spring加载我们的session.xml配置。 我们通过以下配置来做到这一点：

src/main/webapp/WEB-INF/web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spring/*.xml
    </param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

ContextLoaderListener读取contextConfigLocation指定的session.xml配置。

最后，我们需要确保我们的Servlet容器（即Tomcat）为每个请求使用我们的springSessionRepositoryFilter。 以下代码段是最后一步：

src/main/webapp/WEB-INF/web.xml

```
<filter>
    <filter-name>springSessionRepositoryFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
    <filter-name>springSessionRepositoryFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

DelegatingFilterProxy将根据springSessionRepositoryFilter按名称查找一个Bean，并将其转换为Filter。 对于调用DelegatingFilterProxy的每个请求，都将将调用springSessionRepositoryFilter进行处理。

4.3 基于Pivotal GemFire的HttpSession实现 {#httpsession-gemfire}

当Pivotal GemFire与Spring Session一起使用时，Web应用程序的HttpSession可以由GemFire管理的集群实现来替代，并可使用Spring Session的API方便地访问。

使用GemFire管理Spring Sessions的两个最常见的拓扑结构包括：

- Client-Server
- Peer-To-Peer (P2P)

此外，GemFire支持使用WAN功能的站点到站点复制。 配置和使用GemFire的WAN支持与Spring Session无关，超出了本文档的范围。 有关GemFire WAN功能的更多详细信息，请点击此处。

4.3.1 GemFire Client-Server

当使用GemFire作为Spring Session中的提供程序时，Client-Server拓扑可能是更常见的配置首选项，因为与应用程序相比，GemFire服务器将具有显著不同且唯一的JVM堆需求。 使用Client-Server拓扑使应用程序能够独立地管理（例如复制）应用程序状态。

在Client-Server拓扑中，使用Spring Session的应用程序将打开到（远程）GemFire服务器集群的客户端缓存连接，以管理和提供对所有HttpSession状态的一致访问。

您可以使用以下任一配置Client-Server拓扑拓扑：

- 基于Java的配置
- 基于XML的配置

GemFire Client-Server基于java的配置 {#httpsession-gemfire-clientserver-java}

本节介绍如何使用GemFire的Client-Server拓扑来使用基于Java的配置来支持HttpSession。

> HttpSession with GemFire (Client-Server) Sample示例提供了一个关于如何使用Java配置来集成Spring Session和GemFire的工作示例。 您可以阅读下面的集成基本步骤，但是当您与自己的应用程序集成时，建议您遵循GemFire（Client-Server）指南中的详细HttpSession。

spring java配置

添加所需的依赖关系和存储库声明后，我们可以创建我们的Spring配置。 Spring配置负责创建一个使用Spring Session和GemFire支持的实现替换HttpSession的Servlet过滤器。

添加以下Spring配置：

```
@EnableGemFireHttpSession(maxInactiveIntervalInSeconds = 30, poolName = "DEFAULT") 1
public class ClientConfig {

	static final long DEFAULT_WAIT_DURATION = TimeUnit.SECONDS.toMillis(20);

	static final CountDownLatch latch = new CountDownLatch(1);

	static final String DEFAULT_GEMFIRE_LOG_LEVEL = "warning";

	@Bean
	static PropertySourcesPlaceholderConfigurer propertySourcesPlaceholderConfigurer() {
		return new PropertySourcesPlaceholderConfigurer();
	}

	Properties gemfireProperties() {
		Properties gemfireProperties = new Properties();
		gemfireProperties.setProperty("name", applicationName());
		gemfireProperties.setProperty("log-level", logLevel());
		return gemfireProperties;
	}

	String applicationName() {
		return "samples:httpsession-gemfire-clientserver:"
			.concat(getClass().getSimpleName());
	}

	String logLevel() {
		return System.getProperty("sample.httpsession.gemfire.log-level",
			DEFAULT_GEMFIRE_LOG_LEVEL);
	}

	@Bean
	ClientCacheFactoryBean gemfireCache(
			@Value("${spring.session.data.gemfire.port:" + ServerConfig.SERVER_PORT + "}") int port) {

		ClientCacheFactoryBean clientCacheFactory = new ClientCacheFactoryBean();

		clientCacheFactory.setClose(true);
		clientCacheFactory.setProperties(gemfireProperties());

		// GemFire Pool settings
		clientCacheFactory.setKeepAlive(false);
		clientCacheFactory.setPingInterval(TimeUnit.SECONDS.toMillis(5));
		clientCacheFactory.setReadTimeout(2000); // 2 seconds
		clientCacheFactory.setRetryAttempts(1);
		clientCacheFactory.setSubscriptionEnabled(true);
		clientCacheFactory.setThreadLocalConnections(false);

		clientCacheFactory.setServers(Collections.singletonList(
			newConnectionEndpoint(ServerConfig.SERVER_HOST, port)));

		return clientCacheFactory;
	}

	ConnectionEndpoint newConnectionEndpoint(String host, int port) {
		return new ConnectionEndpoint(host, port);
	}

	@Bean
```

```
        BeanPostProcessor gemfireCacheServerReadyBeanPostProcessor() {
                return new BeanPostProcessor() {

                        public Object postProcessBeforeInitialization(Object bean, String beanName)
                                throws BeansException {

                                if ("gemfirePool".equals(beanName)) {
                                        ClientMembership.registerClientMembershipListener(
                                                new ClientMembershipListenerAdapter() {
                                                        @Override
                                                        public void memberJoined(ClientMembershipEvent event) {
                                                                latch.countDown();
                                                        }
                                                });
                                }

                                return bean;
                        }

                        public Object postProcessAfterInitialization(Object bean, String beanName)
                                throws BeansException {

                                if (bean instanceof Pool && "gemfirePool".equals(beanName)) {
                                        try {
                                                Assert.state(latch.await(DEFAULT_WAIT_DURATION, TimeUnit.MILLISECONDS),
                                                        String.format("GemFire Cache Server failed to start on host [%1$s] and port [%2$d]"
,
                                                                ServerConfig.SERVER_HOST, ServerConfig.SERVER_PORT));
                                        }
                                        catch (InterruptedException e) {
                                                Thread.currentThread().interrupt();
                                        }
                                }

                                return bean;
                        }
                };
        }
```

@EnableGemFireHttpSession注释创建一个名为springSessionRepositoryFilter的Spring bean，实现Filter。过滤器是用HttpSession替代由Spring Session和GemFire支持的实现。

接下来，我们注册一个Properties bean，允许我们使用GemFire的System属性来配置GemFire客户端缓存的某些方面。

我们使用属性来配置GemFire ClientCache的实例。

然后，我们配置一个客户端池池，以便在我们的Client / Server拓扑中与GemFire Server通信。在我们的配置中，我们已经使用了明智的设置超时，连接数等。此外，池已配置为直接连接到服务器。从PoolFactory API了解有关各种池配置设置的更多信息。

最后，我们包括一个Spring BeanPostProcessor来阻止客户端，直到我们的GemFire Server启动并运行，监听并接受客户端连接。

gemfireCacheServerReadyBeanPostProcessor是必要的，以便在测试期间以自动方式协调客户端和服务器，但在GemFire群集已经在运行的情况下（例如在生产中）是不必要的。

BeanPostProcessor使用GemFire ClientMembershipListener，当客户端已成功连接到服务器时，将会通知它们。一旦建立连接，监听器释放在PostProcessAfterInitialization回调中BeanPostProcessor将等待的锁存器（直到指定的超时），以阻止客户端。

> 在典型的GemFire部署中，集群中可能包含数百个GemFire数据节点（服务器），客户端更常见地连接到集群中运行的一个或多个GemFire定位器。 定位器将客户端的元数据传递给可用的服务器，负载以及哪些服务器具有客户端感兴趣的数据，这对于单跳直接数据访问特别重要。 在GemFire的用户指南中查看有关客户端/服务器拓扑的更多详细信息。
>
> 有关配置Spring Data GemFire的更多信息，请参阅参考指南。

@EnableGemFireHttpSession注释使开发人员能够使用以下属性来配置Spring Session和GemFire的特定方面：

maxInactiveIntervalInSeconds - 控制HttpSession空闲超时到期（默认为30分钟）。

regionName - 指定用于存储HttpSession状态的GemFire区域的名称（默认为“ClusteredSpringSessions”）。

clientRegionShort - 使用GemFire ClientRegionShortcut（默认为PROXY）指定GemFire的数据管理策略。此属性仅在配置客户端区域时使用。

poolName - 用于将客户端连接到服务器集群的专用GemFire池的名称。该属性仅在应用程序是GemFire缓存客户端时使用。默认为gemfirePool。

serverRegionShort - 使用GemFire RegionShortcut（默认为PARTITION）指定GemFire的数据管理策略。此属性仅在配置服务器区域时使用，或者在采用p2p拓扑时使用。

重要的是要注意，如果客户端区域是PROXY或CACHING_PROXY，则GemFire客户端区域名称必须与服务器区域匹配相同的名称。如果用于存储Spring Sessions的客户端区域是LOCAL，则不需要匹配名称，但是请记住，您的会话状态不会传播到服务器，并且您失去了使用GemFire存储和管理分布式复制会话的所有好处集群中的状态信息。

serverRegionShort在客户机/服务器缓存配置中被忽略，仅在使用对等（P2P）拓扑，更具体地说是GemFire对等体缓存时适用。

服务端配置

我们只涵盖了等式的一边。 我们还需要一个GemFire服务器端，我们的客户端可以与服务器进行通话并且将Session状态发送至服务器端，以进行管理。

在本示例中，GemFire服务端的Java配置如下：
`

```
@EnableGemFireHttpSession(maxInactiveIntervalInSeconds = 30) 1
public class ServerConfig {
    static final int SERVER_PORT = 12480;

    static final String DEFAULT_GEMFIRE_LOG_LEVEL = "warning";
    static final String SERVER_HOST = "localhost";

    @SuppressWarnings("resource")
    public static void main(String[] args) throws IOException { 5
            new AnnotationConfigApplicationContext(ServerConfig.class)
                    .registerShutdownHook();
    }

    @Bean
    static PropertySourcesPlaceholderConfigurer propertyPlaceholderConfigurer() {
            return new PropertySourcesPlaceholderConfigurer();
    }

    Properties gemfireProperties() { 2
            Properties gemfireProperties = new Properties();

            gemfireProperties.setProperty("name", applicationName());
            gemfireProperties.setProperty("mcast-port", "0");
            gemfireProperties.setProperty("log-level", logLevel());
            gemfireProperties.setProperty("jmx-manager", "true");
            gemfireProperties.setProperty("jmx-manager-start", "true");

            return gemfireProperties;
    }

    String applicationName() {
            return "samples:httpsession-gemfire-clientserver:"
                    .concat(getClass().getSimpleName());
    }

    String logLevel() {
            return System.getProperty("sample.httpsession.gemfire.log-level",
                    DEFAULT_GEMFIRE_LOG_LEVEL);
    }

    @Bean
    CacheFactoryBean gemfireCache() { 3
            CacheFactoryBean gemfireCache = new CacheFactoryBean();

            gemfireCache.setClose(true);
            gemfireCache.setProperties(gemfireProperties());

            return gemfireCache;
    }

    @Bean
    CacheServerFactoryBean gemfireCacheServer(Cache gemfireCache,
                    @Value("${spring.session.data.gemfire.port:" + SERVER_PORT + "}") int port) { 4

            CacheServerFactoryBean gemfireCacheServer = new CacheServerFactoryBean();

            gemfireCacheServer.setAutoStartup(true);
            gemfireCacheServer.setBindAddress(SERVER_HOST);
            gemfireCacheServer.setCache(gemfireCache);
            gemfireCacheServer.setHostNameForClients(SERVER_HOST);
            gemfireCacheServer.setMaxTimeBetweenPings(Long.valueOf(TimeUnit.SECONDS.toMillis(60)).intValue());
            gemfireCacheServer.setPort(port);

            return gemfireCacheServer;
    }
```

}
`

1在服务器端也使用@EnableGemFireHttpSession注解来配置Spring Session。 这确保了客户端和服务端的区域名称匹配（在此示例中，我们使用默认

的“ClusteredSpringSessions”）。 我们还将Session的过期时间设置为30秒。 稍后我们将看到如何使用这个过期时间。

2️⃣接下来，我们使用GemFire系统属性配置GemFire服务器，这非常像我们的P2P示例中的配置。 为了使服务端保持独立，我们将mcast-port设置为0并且没有指定locator属性。 GemFire服务端还允许一个使用JMX客户端（例如Gfsh）来连接，但是该JMX必须的系统属性必须是特定于GemFire的。

3️⃣然后，我们创建一个基于GemFire系统属性的GemFire对等缓存实例。

4️⃣我们还设置了在localhost上运行的GemFire CacheServer实例，监听端口12480，准备接受我们的客户端连接。

5️⃣最后，我们将一个main方法声明为从命令行启动作为运行GemFire Server的入口点。

Java Servlet容器初始化

我们的Spring Java配置创建了一个名为springSessionRepositoryFilter的Spring bean，该Bean实现了Filter接口。 springSessionRepositoryFilter bean负责使用基于GemFire支持的Spring Session来替换HttpSession。

为了使springSessionRepositoryFilter过滤器能够做到这一点，Spring需要加载我们的ClientConfig配置类。 还需要确保Servlet容器（即Tomcat）为每个请求使用springSessionRepositoryFilter。 幸运的是，Spring Session提供了一个名为AbstractHttpSessionApplicationInitializer的实用程序类，使这两个步骤都非常容易。

可以参考如下示例：

src/main/java/sample/Initializer.java

public class Initializer extends AbstractHttpSessionApplicationInitializer { 1️⃣

```
    public Initializer() {
            super(ClientConfig.class); 2️⃣
    }
```

}
注意：我们的类（Initializer）的名称并不重要。 重要的是需要扩展AbstractHttpSessionApplicationInitializer类即可。
1️⃣第一步是扩展AbstractHttpSessionApplicationInitializer类。 这确保了一个名为springSessionRepositoryFilter的Spring bean已经注册到我们的Servlet容器并应用于每个请求。

2️⃣AbstractHttpSessionApplicationInitializer还提供了一种便于Spring加载我们的ClientConfig的机制。

GemFire Client-Server基于XML的配置

本节介绍如何使用XML的配置方式来配置基于GemFire的Client-Server拓扑的HttpSession。

> HttpSession with GemFire (Client-Server) using XML Sample提供了一个关于如何使用基于GemFire实现的Spring Session来替换HttpSession的工作示例。 您可以阅读下面的集成基本步骤，但是当与您自己的应用程序集成时，您可以使用XML Guide与GemFire（Client-Server）一起使用详细的HttpSession。

Spring的XML配置

添加所需的依赖关系和存储库声明后，我们可以创建我们的Spring配置。 Spring配置负责创建一个使用Spring Session替换HttpSession的Servlet过滤器。

添加以下Spring配置：

```
1
<context:annotation-config/>
2
<context:property-placeholder location="classpath:META-INF/spring/application.properties"/>
3

<bean class="sample.GemFireCacheServerReadyBeanPostProcessor"/>
4

<util:properties id="gemfireProperties">
        <prop key="log-level">${sample.httpsession.gemfire.log-level:warning}</prop>
</util:properties>
5

<gfe:client-cache properties-ref="gemfireProperties" pool-name="gemfirePool"/>

6
<gfe:pool keep-alive="false"
      ping-interval="5000"
      read-timeout="5000"
      retry-attempts="1"
      subscription-enabled="true"
      thread-local-connections="false">
        <gfe:server host="${application.gemfire.client-server.host}"
            port="${spring.session.data.gemfire.port:${application.gemfire.client-server.port}}"/>
</gfe:pool>

7
<bean class="org.springframework.session.data.gemfire.config.annotation.web.http.GemFireHttpSessionConfiguration"
         p:maxInactiveIntervalInSeconds="30" p:poolName="DEFAULT"/>
```

1使用<context：annotation-config />元素启用Spring注释配置支持，以便在Spring配置中声明的任何使用Spring支持的Spring或Standard Java注释的Spring bean都将被正确配置。

2META-INF / spring / application.properties文件与PropertySourcesPlaceholderConfigurer bean一起使用，以将Spring XML配置元数据中的占位符替换为apprORpriate属性值。

3然后注册“GemFireCacheSeverReadyBeanPostProcessor”，以确定指定主机/端口上的GemFire Server是否正在运行并监听客户端连接，阻止客户端启动，直到服务器可用并准备就绪。

4接下来，我们包括一个Properties bean，以使用GemFire的系统属性来配置GemFire客户端缓存的某些方面。在这种情况下，我们只是从应用程序特定的System属性设置GemFire的日志级别，如果未指定，则默认为警告。

5然后我们创建一个使用我们的gemfireProperties初始化的GemFire ClientCache实例。

6我们配置一个客户端池池，以与客户端/服务器拓扑中的GemFire服务器进行通信。在我们的配置中，我们使用明智的设置超时，连接数等。此外，我们的池已配置为直接连接到服务器。

7最后，注册了GemFireHttpSessionConfiguration以启用Spring Session功能。

> 在典型的GemFire部署中，集群中可能包含数百个GemFire数据节点（服务器），客户端更常见地连接到集群中运行的一个或多个GemFire定位器。 定位器将客户端的元数据传递给可用的服务器，负载以及哪些服务器具有客户端感兴趣的数据，这对于单跳直接数据访问特别重要。 在GemFire的用户指南中查看有关客户端/服务器拓扑的更多详细信息。
>
> 有关配置Spring Data GemFire的更多信息，请参阅参考指南。

服务端配置

我们只涵盖了方程的一边。 我们还需要一个GemFire服务器，我们的客户端可以将会话状态信息与服务器通信并发送到管理中。

在本示例中，我们将使用以下GemFire Server Java配置：

```
        1
        <context:annotation-config/>
        2
        <context:property-placeholder location="classpath:META-INF/spring/application.properties"/>

        3
        <util:properties id="gemfireProperties">
                <prop key="name">GemFireClientServerHttpSessionXmlSample</prop>
                <prop key="mcast-port">0</prop>
                <prop key="log-level">${sample.httpsession.gemfire.log-level:warning}</prop>
                <prop key="jmx-manager">true</prop>
                <prop key="jmx-manager-start">true</prop>
        </util:properties>
        4

        <gfe:cache properties-ref="gemfireProperties"/>
        5

        <gfe:cache-server auto-startup="true"
                      bind-address="${application.gemfire.client-server.host}"
                      host-name-for-clients="${application.gemfire.client-server.host}"
                      port="${spring.session.data.gemfire.port:${application.gemfire.client-server.port}}"/>

        6
        <bean class="org.springframework.session.data.gemfire.config.annotation.web.http.GemFireHttpSessionConfiguration"
                 p:maxInactiveIntervalInSeconds="30"/>
```

1 首先，我们使用<context：annotation-config>元素启用Spring注释配置，以便在Spring配置中声明的任何使用Spring支持的Spring或Standard Java注释的Spring bean都将被正确配置。

2 注册了一个PropertySourcesPlaceholderConfigurer，以便在我们的Spring XML配置元数据中替换META-INF / spring / application.properties文件中的属性值中的占位符。

3 接下来，我们使用GemFire系统属性配置GemFire服务器非常像我们的P2P示例。将mcast-port设置为0并且没有指定locator属性，我们的服务器将是独立的。我们还允许一个JMX客户端（例如Gfsh）使用特定于GemFire的JMX系统属性连接到我们的服务器。

4 然后我们创建一个使用我们的GemFire系统属性初始化的GemFire对等缓存的实例。

5 我们还设置了运行在localhost上的GemFire CacheServer实例，侦听端口11235，准备接受我们的客户端连接。

6 最后，我们通过注册GemFireHttpSessionConfiguration的实例，在客户端上使用相同的Spring Session功能，但我们将会话到期超时设置为30秒。我们稍后会解释这是什么意思。

GemFire Server配置可以通过以下方式进行引导：

```
@Configuration
@ImportResource("META-INF/spring/session-server.xml")
public class Application {

        public static void main(final String[] args) {
                new AnnotationConfigApplicationContext(Application.class)
                        .registerShutdownHook();
        }
}
```

> 而不是使用主要方法的简单Java类，您也可以使用Spring Boot。

@Configuration注释将此Java类定义为使用Spring的注释配置支持的Spring配置元数据的源。

主要来说，配置来自META-INF / spring / session-server.xml文件，这也是Spring样例中没有使用Spring Boot的原因，因为使用XML似乎失败了使用Spring Boot的用途和好处。 但是，本示例将介绍如何使用Spring XML配置GemFire客户端和服务器。

Servlet容器的XML配置

我们的Spring XML配置创建了一个名为springSessionRepositoryFilter的Spring bean，实现了Filter。 springSessionRepositoryFilter bean负责使用Spring Session和GemFire支持的自定义实现替换HttpSession。

为了使我们的过滤器能够做到这一点，我们需要指示Spring加载session-client.xml配置文件。 我们通过以下配置来做到这一点：

src/main/webapp/WEB-INF/web.xml

```
<context-param> <param-name>contextConfigLocation</param-name> <param-value>/WEB-INF/spring/session-client.xml</param-value> </context-p
```

ContextLoaderListener读取contextConfigLocation上下文参数值，并选取我们的session-client.xml配置文件。

最后，我们需要确保我们的Servlet容器（即Tomcat）为每个请求使用我们的springSessionRepositoryFilter。

以下代码段为我们执行最后一步：

```
<filter> <filter-name>springSessionRepositoryFilter</filter-name> <filter-class>org.springframework.web.filter.DelegatingFilterProxy</fi
```

DelegatingFilterProxy将通过springSessionRepositoryFilter的名称查找一个bean，并将其转换为Filter。 对于调用DelegatingFilterProxy的每个请求，将调用

springSessionRepositoryFilter。

4.3.2 GemFire Peer-To-Peer (P2P)

也许较不常见的是使用对等（P2P）拓扑将Spring Session应用程序配置为GemFire群集中的对等成员。在此配置中，Spring Session应用程序将是GemFire群集中的实际数据节点（服务器），而不是以前的缓存客户机。

这种方法的一个优点是应用程序与应用程序的状态（即数据）的接近程度。然而，还有其他有效的方法来完成类似的数据相关计算，例如使用GemFire的功能执行。当GemFire在Spring Session中作为提供商时，任何GemFire的其他功能都可以使用。

P2P对于测试目的以及更小，更集中和自包含的应用程序（如微服务架构中的应用程序）非常有用，并且绝对会提高应用程序的延迟，吞吐量和一致性需求。

您可以使用以下任一方式配置对等（P2P）拓扑：

- 基于Java的配置
- 基于XML的配置

GemFire Peer-To-Peer (P2P) 基于java的配置

本节介绍如何使用GemFire的对等（P2P）拓扑来使用基于Java的配置来支持HttpSession。 使用GemFire（P2P）示例的HttpSession提供了一个关于如何集成Spring Session和GemFire以使用Java配置替换HttpSession的工作示例。 您可以阅读下面的集成基本步骤，但是当您与自己的应用程序集成时，您可以随时使用“GemFire（P2P）指南”中的详细HttpSession。

Spring的java配置

添加所需的依赖关系和存储库声明后，我们可以创建我们的Spring配置。 Spring配置负责创建一个使用Spring Session和GemFire支持的实现替换HttpSession的Servlet过滤器。

添加以下Spring配置：

`@EnableGemFireHttpSession public class Config {

```
    @Bean
    Properties gemfireProperties() {
            Properties gemfireProperties = new Properties();

            gemfireProperties.setProperty("name", "GemFireP2PHttpSessionSample");
            gemfireProperties.setProperty("mcast-port", "0");
            gemfireProperties.setProperty("log-level",
                            System.getProperty("sample.httpsession.gemfire.log-level", "warning"));
            gemfireProperties.setProperty("jmx-manager", "true");
            gemfireProperties.setProperty("jmx-manager-start", "true");

            return gemfireProperties;
    }

    @Bean
    CacheFactoryBean gemfireCache() {
            CacheFactoryBean gemfireCache = new CacheFactoryBean();

            gemfireCache.setClose(true);
            gemfireCache.setProperties(gemfireProperties());

            return gemfireCache;
    }
```

}`

@EnableGemFireHttpSession注释创建一个名为springSessionRepositoryFilter的Spring bean，实现Filter。 这个过滤器是用Hibernate来替代HttpSession的一个实现。 在这种情况下，Spring Session由GemFire支持。 然后，我们使用标准的GemFire系统属性配置GemFire对等缓存。 我们使用name属性给GemFire数据节点一个名称，并将mcast-port设置为0.由于缺少locators属性，该数据节点将是一个独立的服务器。 GemFire的日志级别使用用户可以使用Maven或Gradle运行此示例应用程序时在命令行中指定的应用程序特定的System属性（sample.httpsession.gemfire.log级别）设置（默认为“warning”）。 最后，我们创建一个GemFire对等体缓存的实例，将GemFire嵌入与运行的Spring Session示例应用程序相同的JVM进程。 此外，我们还将此数据节点（服务器）配置为GemFire Manager，并使用特定于GemFire的JMX系统属性，使JMX客户端（例如Gfsh）能够连接到正在运行的数据节点。 有关配置Spring Data GemFire的更多信息，请参阅参考指南。 @EnableGemFireHttpSession注释使开发人员能够使用以下属性来配置Spring Session和GemFire的某些方面：maxInactiveIntervalInSeconds - 控制HttpSession空闲超时到期（默认为30分钟）。

regionName - 指定用于存储HttpSession状态的GemFire区域的名称（默认为“ClusteredSpringSessions”）。

serverRegionShort - 使用GemFire RegionShortcut（默认为PARTITION）指定GemFire数据管理策略。 clientRegionShort在对等缓存配置中被忽略，仅适用于客户机 - 服务器拓扑，更具体地说是使用GemFire客户端缓存。

Java Servlet容器初始化

我们的Spring Java配置创建了一个名为springSessionRepositoryFilter的Spring bean，实现了Filter。 springSessionRepositoryFilter bean负责使用Spring Session和GemFire支持的自定义实现替换HttpSession。

为了使我们的过滤器能够做到这一点，Spring需要加载我们的Config类。 我们还需要确保我们的Servlet容器（即Tomcat）为每个请求使用我们的springSessionRepositoryFilter。 幸运的是，Spring Session提供了一个名为AbstractHttpSessionApplicationInitializer的实用程序类，使这两个步骤都非常容易。

你可以在下面找到一个例子：

src/main/java/sample/Initializer.java

```
public class Initializer extends AbstractHttpSessionApplicationInitializer {

        public Initializer() {
                super(Config.class);
        }
}
```

> 我们的类（Initializer）的名称并不重要。 重要的是我们扩展AbstractHttpSessionApplicationInitializer。

第一步是扩展AbstractHttpSessionApplicationInitializer。 这确保了一个名为springSessionRepositoryFilter的Spring bean已经注册到我们的Servlet容器并用于每个请求。 AbstractHttpSessionApplicationInitializer还提供了一种容易允许Spring加载我们的Config的机制。

GemFire Peer-To-Peer (P2P) 基于XML的配置

本节介绍如何使用GemFire的点对点（P2P）拓扑来使用基于XML的配置来支持HttpSession。

> 使用XML Sample的GemFire（P2P）HttpSession提供了一个关于如何集成Spring Session和GemFire以使用XML配置替换HttpSession的工作示例。 您可以阅读下面的集成基本步骤，但是当与您自己的应用程序集成时，建议您使用XML Guide与GemFire（P2P）一起使用详细的HttpSession。

Spring的XML配置

添加所需的依赖关系和存储库声明后，我们可以创建我们的Spring配置。 Spring配置负责创建一个使用Spring Session和GemFire支持的实现替换HttpSession的Servlet过滤器。 添加以下Spring配置： src/main/webapp/WEB-INF/spring/session.xml

```
<context:annotation-config/>
<context:property-placeholder/>
<bean class="org.springframework.session.data.gemfire.config.annotation.web.http.GemFireHttpSessionConfiguration"/>
<util:properties id="gemfireProperties">
    <prop key="name">GemFireP2PHttpSessionXmlSample</prop>
    <prop key="mcast-port">0</prop>
    <prop key="log-level">${sample.httpsession.gemfire.log-level:warning}</prop>
    <prop key="jmx-manager">true</prop>
    <prop key="jmx-manager-start">true</prop>
</util:properties>
<gfe:cache properties-ref="gemfireProperties" use-bean-factory-locator="false"/>
```

我们使用和GemFireHttpSessionConfiguration的组合，因为Spring Session还没有提供XML命名空间支持（参见gh-104）。这将创建一个名为springSessionRepositoryFilter的Spring bean，它实现Filter。这个过滤器是用Hibernate来替代HttpSession的一个实现。在这种情况下，Spring Session由GemFire支持。 然后，我们使用标准的GemFire系统属性配置GemFire对等缓存。我们使用name属性给GemFire数据节点一个名称，并将mcast-port设置为0.由于缺少locators属性，该数据节点将是一个独立的服务器。 GemFire的日志级别使用用户可以使用Maven或Gradle运行此应用程序时在命令行中指定的应用程序特定的System属性（sample.httpsession.gemfire.log级别）进行设置（默认为“warning”） 。 最后，我们创建一个GemFire对等体缓存的实例，将GemFire嵌入与运行的Spring Session示例应用程序相同的JVM进程。

> 此外，我们还将此数据节点（服务器）配置为GemFire Manager，并使用特定于GemFire的JMX系统属性，使JMX客户端（例如Gfsh）能够连接到正在运行的数据节点。
>
> 有关配置Spring Data GemFire的更多信息，请参阅参考指南。

Servlet容器初始化的XML配置

我们的Spring XML配置创建了一个名为springSessionRepositoryFilter的Spring bean，实现了Filter。 springSessionRepositoryFilter bean负责使用Spring Session和GemFire支持的自定义实现替换HttpSession。

为了使我们的过滤器能够实现其魔力，我们需要指示Spring加载我们的session.xml配置文件。 我们通过以下配置来做到这一点：

src/main/webapp/WEB-INF/web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spring/*.xml
    </param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

ContextLoaderListener读取contextConfigLocation上下文参数值，并选择我们的session.xml配置文件。

最后，我们需要确保我们的Servlet容器（即Tomcat）为每个请求使用我们的springSessionRepositoryFilter。

以下代码段为我们执行最后一步：

src/main/webapp/WEB-INF/web.xml

```
<filter>
    <filter-name>springSessionRepositoryFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
    <filter-name>springSessionRepositoryFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

DelegatingFilterProxy将通过springSessionRepositoryFilter的名称查找一个bean，并将其转换为Filter。 对于调用DelegatingFilterProxy的每个请求，将调用springSessionRepositoryFilter。

4.4 Spring Session-使用JDBC的HttpSession

在使用HttpSession的任何功能之前通过添加一个Servlet过滤器，就可以启用Spring Session，可以通过如下几种方式进行启用：

- 基于Java的配置
- 基于XML的配置
- 基于Spring Boot的配置

4.4.1. 基于Java配置JDBC

本节介绍基于Java配置的方式如何使用关系型数据库支持HttpSession。

> HttpSession JDBC样例提供了一个可执行的样例，这个样例提供了如何基于Java配置整合Spring Session和HttpSession。你可以阅读以下的一些基础步骤，但是当您与自己的应用程序整合时，推荐遵循详细的HttpSession JDBC参考指南。

Spring Java配置

在添加完成必要的依赖之后，我们就可以创建我们自己的配置。Spring配置负责创建一个Servlet过滤器，这个过滤器通过一个使用Spring Session支持的实现去替换HttpSession。添加如下的Spring配置：

```
@EnableJdbcHttpSession 1
public class Config {
    @Bean
    public EmbeddedDatabase dataSource() {
        return new EmbeddedDatabaseBuilder() 2
                .setType(EmbeddedDatabaseType.H2)
                .addScript("org/springframework/session/jdbc/schema-h2.sql").build();
    }

    @Bean
    public PlatformTransactionManager transactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource); 3
    }
}
```

1 @EnableJdbcHttpSession 注解创建了一个实现了Filter接口，命名为springSessionRepositoryFilter的Bean。该过滤器Bean负责使用Spring Session支持的一个实现去替换HttpSession，这个实例中Spring Session由关系型数据库支持。

2 我们创建一个嵌入式数据库H2的实例，使Spring Session连接这个嵌入式数据实例。并且配置Spring Session在应用H2数据库时的SQL脚本，通过SQL脚本来创建数据表。

3 我们创建一个transactionManager去管理前面所创建的数据库的事务。 对于如何配置数据访问的一些相关概念的附加信息，请参考Spring Framework参考文档

Java Servlet容器初始化

我们的Spring配置文件已经创建了一个实现了Filter接口的名为springSessionRepositoryFilter 的Bean。springSessionRepositoryFilter负责使用一个支持Spring Session的实现替换HttpSession。

为了让我们的Filter发挥它的魔力，Spring需要加载我们的Config类。最后我们需要确保每次请求时Servlet容器都使用了springSessionRepositoryFilter。幸运的是，Spring Session提供了一个很便捷的名为AbstractHttpSessionApplicationInitializer的类，使用这个类可以让加载Config类变得非常的容易。参考示例如下：

```
public class Initializer extends AbstractHttpSessionApplicationInitializer { 1

    public Initializer() {
        super(Config.class); 2
    }
}
```

> 我们自己的类（Initializer ）的命名我们并不关心，最重要的是要继承AbstractHttpSessionApplicationInitializer。

1第一步是需要继承AbstractHttpSessionApplicationInitializer。这样可以确保名为springSessionRepositoryFilter 的Spring Bean被注册到Servlet容器中并为每次请求提供处理。
2AbstractHttpSessionApplicationInitializer也提供了一种机制可以非常容易的确保Spring加载Config。

4.4.2. 基于XML配置JDBC

本节介绍基于XML配置的方式如何使用关系型数据库支持HttpSession。

> HttpSession JDBC XML样例提供了一个可执行的样例，这个样例提供了如何基于XML配置整合Spring Session和HttpSession。你可以阅读以下的一些基础步骤，但是当您与自己的应用程序整合时，推荐遵循详细的HttpSession JDBC XML参考指南。

Spring XML配置

添加必要的依赖之后，我们需要创建我们自己的Spring配置。Spring配置主要负责创建一个Spring Session支持的实现去替换HttpSession。Spring配置添加如下：

```
1
<context:annotation-config/>

<bean class="org.springframework.session.jdbc.config.annotation.web.http.JdbcHttpSessionConfiguration"/>
2
<jdbc:embedded-database id="dataSource" database-name="testdb" type="H2">
    <jdbc:script location="classpath:org/springframework/session/jdbc/schema-h2.sql"/>
</jdbc:embedded-database>
3
<bean class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <constructor-arg ref="dataSource"/>
</bean>
```

1我们使用context:annotation-config和JdbcHttpSessionConfiguration主要是因为Spring Session没有提供XML命名空间的支持。这就创建了一个实现了Filter的名为springSessionRepositoryFilter的Spring Bean。此过滤器负责使用Spring Session支持的一个实现去替换HttpSession，这个实例中Spring Session由关系型数据库支持。 2我们创建一个嵌入式数据库H2的实例，使Spring Session连接这个嵌入式数据实例。并且配置Spring Session在应用H2数据库时的SQL脚本，通过SQL脚本来创建数据表。 3我们创建一个transactionManager去管理前面所创建的数据库的事务。

对于如何配置数据访问的一些相关概念的附加信息，请参考Spring Framework参考文档

Servlet容器初始化的XML配置

Spring配置文件创建了一个实现Filter的名为springSessionRepositoryFilter的Bean。springSessionRepositoryFilter负责使用一个支持Spring Session的个性化实现替换HttpSession。

为了让我们的Filter发挥它的魔力，我们需要指示Spring加载我们的session.xml配置文件，我们按照如下配置指示Spring加载session.xml配置文件。

src/main/webapp/WEB-INF/web.xml

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spring/*.xml
    </param-value>
</context-param>
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

ContextLoaderListener读contextConfigLocation并抽出session.xml配置内容。

最后我们需要确保Servlet容器（如Tomcat）的每个请求都使用了springSessionRepositoryFilter，下面的这个代码片段为我们执行了最后一步：

src/main/webapp/WEB-INF/web.xml

```
<filter>
    <filter-name>springSessionRepositoryFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
    <filter-name>springSessionRepositoryFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

DelegatingFilterProxy会根据名称springSessionRepositoryFilter去寻找Bean并将其转化成Filter。对于调用DelegatingFilterProxy的每各请求，将调用springSessionRepositoryFilter。

4.4.3. 基于Spring Boot配置JDBC

本节主要介绍在使用Spring Boot的时候如何使用关系型数据库去支持HttpSession。

> HttpSession JDBC Spring Boot样例提供了一个可执行的样例，这个样例提供了在使用Spring Boot的时候如何整合Spring Session和HttpSession。你可以阅读以下的一些基础步骤，但是当您与自己的应用程序整合时，推荐遵循详细的HttpSession JDBC Spring Boot参考指南。

Spring Boot配置

添加所需的依赖关系后，我们可以创建我们的Spring配置。 Spring配置主要负责创建一个Spring Session支持的实现去替换HttpSession。Spring配置添加如下：

```
@EnableJdbcHttpSession public class HttpSessionConfig { }
```

1 @EnableJdbcHttpSession注解创建一个Spring Bean，名称为springSessionRepositoryFilter，该Bean实现了Filter接口。 这个过滤器是负责替换由Spring Session支持的HttpSession实现的过程。 在这种情况下，Spring Session由关系数据库支持。

配置DataSource

Spring Boot会自动创建DataSource连接Spring Session和嵌入的H2数据库实例。在生产环境中，你需要确保更新你的配置到你的关系型数据库中。例如，你需要在application.properties包含下列内容：

```
spring.datasource.url=jdbc:postgresql://localhost:5432/myapp
spring.datasource.username=myapp
spring.datasource.password=secret
```

更多内容，请参考Spring Boot参考文档的配置数据源部分。

Servlet 容器初始化

Spring Boot配置文件创建了一个实现了Filter接口的名为springSessionRepositoryFilter的Bean，springSessionRepositoryFilter负责使用一个支持Spring Session的个性化实现替换HttpSession。

为了让我们的Filter发挥它的魔力，Spring需要加载我们的Config类。最后我们需要确保每次请求Servlet容器都使用了springSessionRepositoryFilter。幸运的是Spring Boot已经帮我们实现了。

4.5 使用Mongo的HttpSession

在使用HttpSession的任何东西之前，通过添加Servlet过滤器来启用使用HttpSession的Spring Session支持。

本节介绍基于Java的配置方式如何使用Mongo来支持HttpSession。

> HttpSession Mongo Sample提供了一个关于如何使用Java配置集成Spring Session和HttpSession的工作示例。 您可以阅读下面的集成基本步骤，但是当您与自己的应用程序集成时，建议您遵循详细的HttpSession指南。 所有您需要做的是添加以下Spring配置：

```
@EnableMongoHttpSession 1
public class HttpSessionConfig {

        @Bean
        public JdkMongoSessionConverter jdkMongoSessionConverter() {
                return new JdkMongoSessionConverter(); 2
        }
}
```

1 @EnableMongoHttpSession注释创建一个Spring Bean，名称为springSessionRepositoryFilter，实现Filter接口。 这个过滤器负责将Http Session替换为Spring Session。 在这种情况下，Spring Session由Mongo支持。

2 我们显式地配置JdkMongoSessionConverter，因为Spring Security的对象不能使用Jackson自动保留（默认情况下，如果Jackson位于类路径上）。

4.5.1 Session序列化机制

为了能够在MongoDB中保留Session对象，我们需要提供序列化/反序列化机制。 根据您的类路径，Spring Session将选择两个内置转换器之一：

- 当ObjectMapper类可用时，JacksonMongoSessionConverter，或
- 否则JdkMongoSessionConverter。

JacksonMongoSessionConverter

这个机制使用Jackson将Session对象序列化成JSON。 当在类路径上检测到Jackson时，并且用户尚未明确注册AbstractMongoSessionConverter Bean时，JacksonMongoSessionConverter将自动被启用。 如果您想提供定制的Jackson模块，您可以通过明确注册JacksonMongoSessionConverter：

```
@Configuration
@EnableMongoHttpSession
public class MongoJacksonSessionConfiguration {

        @Bean
        public AbstractMongoSessionConverter mongoSessionConverter() {
                return new JacksonMongoSessionConverter(getJacksonModules());
        }

        public Iterable<Module> getJacksonModules() {
                return Collections.<Module>singletonList(new MyJacksonModule());
        }
}
```

JdkMongoSessionConverter

JdkMongoSessionConverter使用标准Java序列化机制将Session属性以二进制形式映射到MongoDB。 然而，标准Session元素（如id，访问时间等）仍然被写为一个普通的Mongo对象，可以无需额外的工作就可以读取和查询。如果Jackson库不在类路径，并且没有明确的AbstractMongoSessionConverter Bean被定义， 则JdkMongoSessionConverter被使用。 您可以通过将其定义为Bean来显式注册JdkMongoSessionConverter。

```
@Configuration
@EnableMongoHttpSession
public class MongoJdkSessionConfiguration {

        @Bean
        public AbstractMongoSessionConverter mongoSessionConverter() {
                return new JdkMongoSessionConverter();
        }
}
```

JdkMongoSessionConverter还有一个带有Serializer和Deserializer两个参数的构造函数，允许您传递自定义实现，这在要使用非默认类加载器时尤为重要。

使用自定义转换器

您可以通过扩展AbstractMongoSessionConverter类来创建自己的Session转换器。 该实现将用于对您的对象进行序列化，反序列化和提供访问Session的查询。

4.6 使用Hazelcast的HttpSession

在使用HttpSession的任何东西之前，通过添加Servlet过滤器来启用使用HttpSession的Spring Session。

本节介绍如何使用Hazelcast基于Java的配置来返回HttpSession。 Hazelcast Spring Sample提供了一个关于如何使用Java配置集成Spring Session和HttpSession的工作示例。 您可以阅读下面的集成基本步骤，但是当与您自己的应用程序集成时，您可以遵循详细的“Hazelcast Spring指南”。

Spring配置

添加所需的依赖关系后，我们可以创建我们的Spring配置。 Spring配置负责创建一个使用Spring Session替换HttpSession的Servlet过滤器。 添加以下Spring配置：

```
@EnableHazelcastHttpSession 1
@Configuration
public class HazelcastHttpSessionConfig {

	@Bean
	public HazelcastInstance hazelcastInstance() {
		MapAttributeConfig attributeConfig = new MapAttributeConfig()
				.setName(HazelcastSessionRepository.PRINCIPAL_NAME_ATTRIBUTE)
				.setExtractor(PrincipalNameExtractor.class.getName());

		Config config = new Config();

		config.getMapConfig("spring:session:sessions") 2
				.addMapAttributeConfig(attributeConfig)
				.addMapIndexConfig(new MapIndexConfig(
						HazelcastSessionRepository.PRINCIPAL_NAME_ATTRIBUTE, false));

		return Hazelcast.newHazelcastInstance(config); 3
	}

}
```

1 @EnableHazelcastHttpSession注解创建一个名为springSessionRepositoryFilter的Spring Bean，该Bean实现Filter接口。 该过滤器是负责由Spring Session替换HttpSession的过程。 在这种情况下，Spring Session由Hazelcast提供支持。

2 为了支持按主体名称索引Session，需要注册适当的ValueExtractor。 Spring Session为此提供了PrincipalNameExtractor。

3 我们创建一个将Spring Session连接到Hazelcast的HazelcastInstance。 默认情况下，应用程序启动并连接到一个嵌入式的Hazelcast实例。 有关配置Hazelcast的更多信息，请参阅参考文档。

Servlet容器初始化

我们的Spring Configuration创建了一个实现了Filter接口，并且名为springSessionRepositoryFilter的Spring Bean。 springSessionRepositoryFilter bean负责使用Spring Session支持的自定义实现替换HttpSession。 为了使我们的过滤器能够做到这一点，Spring需要加载我们的SessionConfig类。 由于我们的应用程序已经使用SecurityInitializer类加载了Spring配置，所以我们可以简单地添加我们的SessionConfig类。 src/main/java/sample/SecurityInitializer.java

```
public class SecurityInitializer extends AbstractSecurityWebApplicationInitializer {

	public SecurityInitializer() {
		super(SecurityConfig.class, SessionConfig.class);
	}
}
```

最后，我们需要确保我们的Servlet容器（即Tomcat）为每个请求使用我们的springSessionRepositoryFilter。 在Spring Security的springSecurityFilterChain之前调用Spring Session的springSessionRepositoryFilter是非常重要的。 这样可以确保Spring Security使用的HttpSession由Spring Session支持。 幸运的是，Spring Session提供了一个名为AbstractHttpSessionApplicationInitializer的实用程序类，使其非常简单。 你可以在下面找到一个例子： src/main/java/sample/Initializer.java

```
public class Initializer extends AbstractHttpSessionApplicationInitializer {

}
```

> 我们的类（Initializer）的名称并不重要。 重要的是我们扩展AbstractHttpSessionApplicationInitializer。 通过扩展AbstractHttpSessionApplicationInitializer，我们确保在Spring Security的springSecurityFilterChain之前，Spring Bean的名称为springSessionRepositoryFilter，在Servlet容器中注册了每个请求。

4.7 HttpSession集成的工作原理

幸运的是，HttpSession和HttpServletRequest（用于获取HttpSession的API）都是接口。 这意味着我们可以为每个这些API提供我们自己的实现。

> 本节介绍Spring Session如何提供与HttpSession的透明集成。 目的是让用户可以了解底层的原理。 此功能已经集成，您不需要自己实现此逻辑。

首先我们创建一个自定义的HttpServletRequest，返回HttpSession的自定义实现。 它看起来像下面这样：

```
public class SessionRepositoryRequestWrapper extends HttpServletRequestWrapper {

        public SessionRepositoryRequestWrapper(HttpServletRequest original) {
                super(original);
        }

        public HttpSession getSession() {
                return getSession(true);
        }

        public HttpSession getSession(boolean createNew) {
                // create an HttpSession implementation from Spring Session
        }

        // ... other methods delegate to the original HttpServletRequest ...
}
```

任何返回HttpSession的方法都将被覆盖。 所有其他方法都由HttpServletRequestWrapper实现，只需委托给原来的HttpServletRequest实现。

我们使用名为SessionRepositoryFilter的Servlet Filter来替换HttpServletRequest实现。 伪代码可以在下面找到：

```
public class SessionRepositoryFilter implements Filter {

        public doFilter(ServletRequest request, ServletResponse response, FilterChain chain) {
                HttpServletRequest httpRequest = (HttpServletRequest) request;
                SessionRepositoryRequestWrapper customRequest =
                        new SessionRepositoryRequestWrapper(httpRequest);

                chain.doFilter(customRequest, response, chain);
        }

        // ...
}
```

通过将自定义的HttpServletRequest实现传递给FilterChain，我们确保在过滤器使用自定义HttpSession实现后调用的任何内容。 这突出了为什么重要的是Spring Session的SessionRepositoryFilter必须放在与HttpSession交互的任何内容之前。

4.8 单浏览器中的多个HttpSession

Spring Session能够在单个浏览器实例中支持多个Session。 这样可以支持在同一浏览器实例（例如Google帐户）中进行多个用户进行身份验证。

> “Manage Multiple Users Guide”提供了在同一浏览器实例中管理多个用户的完整工作示例。 您可以按照以下基本步骤进行集成，但是当与您自己的应用程序集成时，建议您遵循详细的“管理多用户指南”。

我们来看看Spring Session如何跟踪多个会话。

管理单个Session

Spring Session通过向名为SESSION的cookie添加值来跟踪HttpSession。 例如，SESSION cookie可能具有以下值：

```
7e8383a4-082c-4ffe-a4bc-c40fd3363c5e
```

添加一个Session

我们可以通过请求包含特殊参数的URL来添加另一个会话。 默认情况下，参数名称为_s。 例如，以下URL将创建一个新的Session：

HTTP://localhost:8080/_s=1

> 参数值不表示实际的会话ID。 这很重要，因为我们不希望允许客户端确定会话ID以避免Session固定攻击。 另外，我们不希望会话ID被作为查询参数发送。 记住敏感信息只能作为Header或请求的Body传送。

除了自己创建URL，我们可以利用HttpSessionManager来创建一个Session。 我们可以使用以下方法从HttpServletRequest获取HttpSessionManager：
src/main/java/sample/UserAccountsFilter.java

```
HttpSessionManager sessionManager = (HttpSessionManager) httpRequest
        .getAttribute(HttpSessionManager.class.getName());
```

我们现在可以使用它创建一个URL来添加另一个Session。 src/main/java/sample/UserAccountsFilter.java

```
String addAlias = unauthenticatedAlias == null ? 1
        sessionManager.getNewSessionAlias(httpRequest)
        : 2
        unauthenticatedAlias; 3
String addAccountUrl = sessionManager.encodeURL(contextPath, addAlias); 4
```

1 我们有一个名为unauthenticatedAlias的变量。 该值是指向现有未认证Session的别名。 如果不存在此Session，则该值为null。 这样可以确保我们使用现有的未经身份验证的Session，而不是创建新的Session。

2 如果我们所有的Session都已经与用户关联，我们将创建一个新的Session别名。

3 如果存在与用户没有关联的现有Session，则会使用其Session别名。

4 最后，我们创建添加帐户URL。 该URL包含一个Session别名，它指向一个现有的未认证Session，或者是一个未被使用的别名，从而发出信号，创建与该别名关联的新Session。 现在我们的SESSION cookie看起来像这样：

```
0 7e8383a4-082c-4ffe-a4bc-c40fd3363c5e 1 1d526d4a-c462-45a4-93d9-84a39b6d44ad
```

这样：

- sessionID为 7e8383a4-082c-4ffe-a4bc-c40fd3363c5e
- 此Session的别名为0.例如，如果URL为http://localhost:8080 /？ _s = 0，则将使用此别名。
- 这是默认Session。 这意味着如果没有指定Session别名，则使用此Session。 例如，如果URL是http://localhost:8080/将使用此Session。
- Sessionid 1d526d4a-c462-45a4-93d9-84a39b6d44ad
- 此Session的别名为1.如果Session别名为1，则使用此Session。 例如，如果URL是http：// localhost:8080 /？ _s = 1，则使用此别名。

自动包含Session别名的encodeURL

在URL中指定Session别名的好处是，我们可以使用不同的活动Session打开多个选项卡。 不好的是，我们需要在我们应用程序的每个URL中包含Session别名。 幸运的是，Spring Session会通过HttpServletResponse # encodeURL（java.lang.String）方法自动在URL中添加Session别名，

这意味着如果您使用标准标签库，Session别名将自动包含在URL中。 例如，如果我们正在使用具有1的别名的会话，那么以下内容： src/main/webapp/index.jsp

```
<c:url value="/link.jsp" var="linkUrl"/>
<a id="navLink" href="${linkUrl}">Link</a>
```

输出的链接如下：

```
<a id="navLink" href="/link.jsp?_s=1">Link</a>
```

4.9 HttpSession & RESTful APIs

Spring session可以通过允许在标题中提供会话来使用RESTful API。

> REST示例提供了一个关于如何在REST应用程序中使用Spring Session来支持http请求头进行身份验证的工作示例。 您可以按照以下基本步骤进行集成，但在与自己的应用程序集成时，建议您遵循详细的REST指南。

spring配置

添加所需的依赖关系后，我们可以创建我们的Spring配置。 Spring配置负责创建一个使用Spring Session支持的实现替换HttpSession实现的Servlet过滤器。 添加以下Spring配置：

```
@Configuration
@EnableRedisHttpSession ①
public class HttpSessionConfig {

        @Bean
        public LettuceConnectionFactory connectionFactory() {
                return new LettuceConnectionFactory(); ②
        }

        @Bean
        public HttpSessionStrategy httpSessionStrategy() {
                return new HeaderHttpSessionStrategy(); ③
        }
}
```

①@EnableRedisHttpSession注释创建一个名为springSessionRepositoryFilter的Spring Bean，该实例实现了Filter。 过滤器是负责替换由Spring Session支持的HttpSession实现的过程。 在这种情况下，Spring Session由Redis支持。

②我们创建一个将Spring Session连接到Redis Server的RedisConnectionFactory。 我们配置默认端口为（6379）。有关配置，请参考Spring Data Redis的参考文档。

③实现自定义Spring Session的HttpSession集成，使用HTTP请求来传输当前会话信息而不是Cookie。

Servlet容器的初始化

我们的Spring Configuration创建了一个名为springSessionRepositoryFilter的Spring Bean，改Bean实现了Filter接口。 springSessionRepositoryFilter bean负责使用Spring Session支持的自定义实现来替换HttpSession。

为了使我们的过滤器能够做到这一点，Spring需要加载我们的Config类。 我们在Spring MvcInitializer中提供了如下配置：

src/main/java/sample/mvc/MvcInitializer.java

```
@Override
protected Class<?>[] getRootConfigClasses() {
    return new Class[] { SecurityConfig.class, HttpSessionConfig.class };
}
```

最后，我们需要确保我们的Servlet容器（即Tomcat）为每个请求使用我们的springSessionRepositoryFilter。 幸运的是，Spring Session提供了一个名为AbstractHttpSessionApplicationInitializer的实用程序类，使其非常简单。 只需使用默认构造函数扩展类，如下所示：

src/main/java/sample/Initializer.java

```
public class Initializer extends AbstractHttpSessionApplicationInitializer {

}
```

> 我们的类（Initializer）的名称并不重要。 重要的是我们扩展AbstractHttpSessionApplicationInitializer。

4.10 HttpSession & RESTful APIs

Spring Session通过声明SessionEventHttpSessionListenerAdapter将SessionDestroyedEvent和SessionCreatedEvent转换为HttpSessionEvent来支持HttpSessionListener。 要使用HttpSessionListener，您需要：

- 确保您的SessionRepository实现并配置为触发SessionDestroyedEvent和SessionCreatedEvent。
- 将SessionEventHttpSessionListenerAdapter配置为Spring bean。
- 将每个HttpSessionListener注入到SessionEventHttpSessionListenerAdapter中

如果您正在使用本文档中提到的HttpSession with Redis，那么您需要做的就是将每个HttpSessionListener注册为一个bean。 例如，假设您要支持Spring Security的并发控制，并且需要使用HttpSessionEventPublisher，您可以简单地将HttpSessionEventPublisher添加为一个bean。 在Java配置中，可能如下所示：

```
@Configuration
@EnableRedisHttpSession
public class RedisHttpSessionConfig {

        @Bean
        public HttpSessionEventPublisher httpSessionEventPublisher() {
                return new HttpSessionEventPublisher();
        }

        // ...
}
```

采用XML的配置方式如下：

```
<bean class="org.springframework.security.web.session.HttpSessionEventPublisher"/>
```

5.WebSocket的集成

5.1 为什么选择Spring Session & WebSockets? {#websocket-why}

那么为什么在使用WebSockets时需要Spring Session呢?

考虑一个通过HTTP请求完成大部分工作的电子邮件应用程序。以及通过WebSocket API实现的聊天应用程序。 如果一个用户正在和某人进行积极的聊天，那么我们不应该让HttpSession超时，因为这样会导致很差的用户体验。 因此，这正是JSR-356解决的问题。

另一个问题是，根据JSR-356，如果HttpSession超时，则使用该HttpSession创建的任何WebSocket以及登陆认证的用户都应被强制关闭。 这意味着如果我们正在应用程序中聊天，如果没有操作HttpSession，那么我们也将被断开聊天对话!

5.2 WebSocket中应用Spring Session

WebSocket Sample提供了有关如何将Spring Session与WebSockets集成的示例。 您可以按照以下基本步骤进行集成，但是当与您自己的应用程序集成时，建议您遵循详细的“WebSocket指南”：

5.2.1 HttpSession集成

在使用WebSocket集成之前，您应该确保HttpSession正常工作。

5.2.2 Spring配置

在一个典型的Spring WebSocket应用程序中，用户将扩展AbstractWebSocketMessageBrokerConfigurer。 例如，配置可能如下所示：

```
@Configuration
@EnableScheduling
@EnableWebSocketMessageBroker
public class WebSocketConfig extends AbstractWebSocketMessageBrokerConfigurer {

        public void registerStompEndpoints(StompEndpointRegistry registry) {
                registry.addEndpoint("/messages").withSockJS();
        }

        @Override
        public void configureMessageBroker(MessageBrokerRegistry registry) {
                registry.enableSimpleBroker("/queue/", "/topic/");
                registry.setApplicationDestinationPrefixes("/app");
        }
}
```

我们可以轻松地更新您的配置以使用Spring Session WebSocket支持。 例如：

```
@Configuration
@EnableScheduling
@EnableWebSocketMessageBroker
public class WebSocketConfig
                extends AbstractSessionWebSocketMessageBrokerConfigurer<ExpiringSession> { 1

        protected void configureStompEndpoints(StompEndpointRegistry registry) { 2
                registry.addEndpoint("/messages").withSockJS();
        }

        public void configureMessageBroker(MessageBrokerRegistry registry) {
                registry.enableSimpleBroker("/queue/", "/topic/");
                registry.setApplicationDestinationPrefixes("/app");
        }
}
```

在WebSocket与Spring Session集成中，我们只需要改变两件事情：

1️⃣ WebSocket的配置累集成AbstractSessionWebSocketMessageBrokerConfigurer而不是AbstractWebSocketMessageBrokerConfigurer我们扩展

2️⃣ 重命名registerStompEndpoints方法来配置StompEndpoints

BackgroundSessionWebSocketMessageBrokerConfigurer在幕后做什么？

- WebSocketConnectHandlerDecoratorFactory作为WebSocketHandlerDecoratorFactory添加到WebSocketTransportRegistration。这样可以确保包含WebSocketSession的SessionConnectEvent被触发。当Spring Session终止时，WebSocketSession必须终止任何WebSocket连接。
- SessionRepositoryMessageInterceptor作为HandshakeInterceptor添加到每个StompWebSocketEndpointRegistration中。这样可以确保将Session添加到WebSocket属性，以便更新上次访问的时间。
- 将SessionRepositoryMessageInterceptor作为ChannelInterceptor添加到我们的入站ChannelRegistration中。这样可以确保每次收到入站邮件时，我们的Spring Session的上次访问时间都将被更新。
- WebSocketRegistryListener被创建为一个Spring Bean。这样可以确保我们将所有Session ID映射到相应的WebSocket连接。通过维护此映射，当Spring Session（HttpSession）终止时，我们可以关闭所有的WebSocket连接。

6.Spring Security的集成

Spring Session 支持与Spring Security的集成。

6.1 Spring Security"记住我"功能的支持

Spring Session提供了与Spring Security记住我功能的支持，这包括：

- 延长session的过期时间
- 确保session cookie 的过期时间为Integer.MAX_VALUE。cookie过期时间被设置为最大可能的值，因为仅在创建session时设置cookie。 如果将cookie的过期时间与session的过

期时间设置为相同的值，则当用户使用该session时，session将被更新，但是cookie的过期时间不会被更新，从而导致cookie的过期时间固定。

基于Java的配置方式将Spring Session集成到Spring Security中，请使用以下内容作为参考：

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        // ... additional configuration ...
        .rememberMe()
            .rememberMeServices(rememberMeServices());
}

@Bean
RememberMeServices rememberMeServices() {
    SpringSessionRememberMeServices rememberMeServices =
            new SpringSessionRememberMeServices();
    // optionally customize
    rememberMeServices.setAlwaysRemember(true);
    return rememberMeServices;
}
```

基于XML的配置方式如下：

```
<security:http>
    <!-- ... -->
    <security:form-login />
    <security:remember-me services-ref="rememberMeServices"/>
</security:http>

<bean id="rememberMeServices"
    class="org.springframework.session.security.web.authentication.SpringSessionRememberMeServices"
    p:alwaysRemember="true"/>
```

6.2 Spring Security对并发session的控制

Spring Session支持Spring Security的并发session的控制。这允许限制单个用户可以并发的活动会话数，但与默认的Spring Security支持不同，这也可以在集群环境中使用。 这通过提供Spring Security的SessionRegistry接口的自定义实现来实现。

当使用Spring Security的Java配置DSL时，可以通过SessionManagementConfigurer配置自定义SessionRegistry，如下所示：

```
@Configuration
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

        @Autowired
        FindByIndexNameSessionRepository<ExpiringSession> sessionRepository;

        @Override
        protected void configure(HttpSecurity http) throws Exception {
                http
                        // other config goes here...
                        .sessionManagement()
                                .maximumSessions(2)
                                .sessionRegistry(sessionRegistry());
        }

        @Bean
        SpringSessionBackedSessionRegistry sessionRegistry() {
                return new SpringSessionBackedSessionRegistry(this.sessionRepository);
        }
}
```

这假设您还配置了Spring Session来提供一个返回ExpiringSession实例的FindByIndexNameSessionRepository。

使用XML配置时，会看起来像这样：

```
<security:http>
    <!-- other config goes here... -->
    <security:session-management>
        <security:concurrency-control max-sessions="2" session-registry-ref="sessionRegistry"/>
    </security:session-management>
</security:http>

<bean id="sessionRegistry"
      class="org.springframework.session.security.SpringSessionBackedSessionRegistry">
    <constructor-arg ref="sessionRepository"/>
</bean>
```

这假设你的Spring Session SessionRegistry bean被称为sessionRegistry，它是所有SpringHttpSessionConfiguration子类使用的名称，除了用于MongoDB的名称之外：它被称为mongoSessionRepository。

6.3 局限

Spring Session实现的Spring Security的SessionRegistry接口不支持getAllPrincipals方法，因为这个信息无法使用Spring Session进行检索。 Spring Security不会调用此方法，因此这仅影响访问SessionRegistry本身的应用程序。

7.API文档

您可以在线浏览完整的Javadoc。 关键API如下所述：

7.1 Session

session是key-value的集合，可以看作是简化的Map。

session的典型用法如下：

```
public class RepositoryDemo<S extends Session> {
    private SessionRepository<S> repository; ①

    public void demo() {
        S toSave = this.repository.createSession(); ②

        ③
        User rwinch = new User("rwinch");
        toSave.setAttribute(ATTR_USER, rwinch);

        this.repository.save(toSave); ④

        S session = this.repository.getSession(toSave.getId());

        ⑥
        User user = session.getAttribute(ATTR_USER);⑤
        assertThat(user).isEqualTo(rwinch);
    }

    // ... setter methods ...
}
```

①创建一个泛型的SessionRepository实例，S代表Session的子类，泛型的具体类型通过SessionRepository来获取。

②通过SessionRepository创建一个新的Session实例，将此实例赋值给泛型S声明的变量。

③操作Session，这里我们将一个User实例保存到Session中。

④保存Session，泛型S的作用就体现在这里. SessionRepository只允许保存使用相同SessionRepository创建或获取的Session实例。 这允许SessionRepository进行特定的优化（即仅写入已经改变的属性）。

⑤从SessionRepository获取Session实例。

⑥从Session中获取持久化的User，这里不需要强制类型转换。

7.2 ExpiringSession

ExpiresSession继承Session，扩展了与Session过期时间相关的属性。 如果程序不需要与Session过期时间进行交互，则推荐使用更简单的Session API。

ExpiringSession的典型用法如下：

```
public class ExpiringRepositoryDemo<S extends ExpiringSession> {
    private SessionRepository<S> repository; ①

    public void demo() {
        S toSave = this.repository.createSession(); ②
        // ...
        toSave.setMaxInactiveIntervalInSeconds(30); ③

        this.repository.save(toSave); ④

        S session = this.repository.getSession(toSave.getId()); ⑤
        // ...
    }

    // ... setter methods ...
}
```

①创建一个泛型的SessionRepository实例，S代表ExpiringSession的子类，泛型的具体类型通过SessionRepository来获取。

②通过SessionRepository创建一个新的ExpiringSession实例，将此实例赋值给泛型S声明的变量。

③操作Session，这里我们演示如何在Session过期之前更新Session的过期时间。

④保存Session，泛型S的作用就体现在这里. SessionRepository只允许保存使用相同SessionRepository创建或获取的Session实例。 这允许SessionRepository进行特定的优化（即仅写入已经改变的属性）。当ExpiringSession被保存时，最后访问时间会自动更新。

⑤从SessionRepository获取Session实例，如果Session已经过期，则结果为Null。

7.3 SessionRepository

SessionRepository负责创建、获取和持久化Session实例。

开发人员尽量避免直接与SessionRepository或Session进行交互。 相反，开发人员应该倾向于通过与HttpSession和WebSocket集成的方式来间接地与SessionRepository和Session进行交互。

7.4 FindByIndexNameSessionRepository

Spring Session中用于操作Session的最基本的API是SessionRepository。 SessionRepository力求简单，因此可以轻松提供具有基本功能的其他实现。

一些SessionRepository实现也可以选择实现FindByIndexNameSessionRepository。 例如，Spring的Redis支持，就实现了FindByIndexNameSessionRepository接口。

FindByIndexNameSessionRepository添加一个方法来查找特定用户的所有Session。 这是通过使用FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME为key的会话属性填充用户名来完成的。 开发人员负责添加属性，因为Spring Session不知道正在使用的身份验证机制。 可以参考下面的例子：

```
String username = "username";
this.session.setAttribute(
        FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME, username);
```

> FindByIndexNameSessionRepository的一些实现将提供钩子来自动索引其他会话属性。 例如，许多实现将自动确保当前的Spring Security用户名使用索引名称FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME进行索引。

Session被索引后，可以通过如下方式找到：

```
String username = "username";
Map<String, Session> sessionIdToSession = this.sessionRepository
        .findByIndexNameAndIndexValue(
                FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME,
                username);
```

7.5 EnableSpringHttpSession

@EnableSpringHttpSession注释可以添加到@Configuration注解的配置类上，用于暴露一个名称为“springSessionRepositoryFilter“

的SessionRepositoryFilter。 为了使用这个注释，必须提供SessionRepository bean。 例如：

```
@EnableSpringHttpSession
@Configuration
public class SpringHttpSessionConfig {
        @Bean
        public MapSessionRepository sessionRepository() {
                return new MapSessionRepository();
        }
}
```

注意，在Spring Session的基础架构中并没有提供开箱即用的Session过期配置。 这是因为Session的过期时间高度依赖于具体的实现。 这意味着如果您需要清理已过期的Session，需要自己实现清理过期的Session。

7.6 edisOperationsSessionRepository

RedisOperationsSessionRepository是基于Spring Data的RedisOperations实现的SessionRepository。 在Web环境中，这通常与SessionRepositoryFilter结合使用。 该实现通过SessionMessageListener支持SessionDestroyedEvent和SessionCreatedEvent。

7.6.1 初始化一个RedisOperationsSessionRepository实例

```
LettuceConnectionFactory factory = new LettuceConnectionFactory();
SessionRepository<? extends ExpiringSession> repository = new RedisOperationsSessionRepository(
        factory);
```

有关如何创建RedisConnectionFactory的其他信息，请参考Spring Data Redis参考。

7.6.2 EnableRedisHttpSession

在Web环境中，创建RedisOperationsSessionRepository的最简单方法是使用@EnableRedisHttpSession注解。 Samples and Guides中可以找到完整的示例用法，您可以使用以下属性来自定义配置：

- maxInactiveIntervalInSeconds - Session过期时间
- redisNamespace - 允许为Session配置应用程序特定的命名空间。 Redis的Key和channel ID将以spring:session:<redisNamespace>：前缀开头。
- redisFlushMode - 允许指定何时将数据写入Redis。 默认情况下，只有在SessionRepository上调用save时，才将数据写入Redis。 RedisFlushMode.IMMEDIATE模式将尽快将数据写入Redis。

自定义RedisSerializer {#custom-redisserializer}

通过创建一个名为springSessionDefaultRedisSerializer，并且实现RedisSerializer <Object>接口的Bean来自定义序列化。

7.6.3 Redis TaskExecutor

RedisOperationsSessionRepository使用RedisMessageListenerContainer从redis接收事件。 您可以通过创建名为springSessionRedisTaskExecutor的Bean和/或springSessionRedisSubscriptionExecutor来自定义这些事件的分派方式。 有关配置redis任务执行程序的更多详细信息，请参见此处。

7.6.4 存储细节

以下部分概述了Redis如何针对每个操作进行更新。 创建新Session的示例如下。 后续部分将详细介绍。

```
HMSET spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe creationTime 1404360000000 \
    maxInactiveInterval 1800 \
    lastAccessedTime 1404360000000 \
    sessionAttr:attrName someAttrValue \
    sessionAttr2:attrName someAttrValue2
EXPIRE spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe 2100
APPEND spring:session:sessions:expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe ""
EXPIRE spring:session:sessions:expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe 1800
SADD spring:session:expirations:1439245080000 expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe
EXPIRE spring:session:expirations1439245080000 2100
```

存储Session

每个Session都以Hash形式存储在Redis中。 使用HMSET命令设置和更新每个Session。 下面可以看到每个Session如何存储的一个例子。

```
HMSET spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe creationTime 1404360000000 \
    maxInactiveInterval 1800 \
    lastAccessedTime 1404360000000 \
    sessionAttr:attrName someAttrValue \
    sessionAttr2:attrName someAttrValue2
```

在此示例中，Session的各部分分别为：

sessionid是：33fdd1b6-b496-4b33-9f7d-df96679d32fe

session的创建时间是1404360000000，这是一个从1/1/1970 GMT开始的一个时间戳。

session的过期时间是1800秒（30分钟）

session的最后访问时间是1404360000000，这是一个从1/1/1970 GMT开始的一个时间戳。

session有两个属性。第一个属性的名称是"attrName"，值是"someAttrValue"，第二个属性的名称是"attrName2"，值是"someAttrValue2"。

Session优化写入值

由RedisOperationsSessionRepository管理的Session实例跟踪已更改的属性，并只更新这些属性。 这意味着如果一个属性被写入一次并且读取很多次，我们只需要写入该属性一次。 例如，假设前面的会话属性“sessionAttr2”已更新。 保存后将执行以下操作：

```
HMSET spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe sessionAttr:attrName2 newValue
```

Session过期时间 {#api-redisoperationssessionrepository-expiration}

每个Session的过期时间用ExpiringSession.getMaxInactiveInterval()来获取，通过EXPIRE命令来更新过期时间。 例如：

```
EXPIRE spring:session:sessions:33fdd1b6-b496-4b33-9f7d-df96679d32fe 2100
```

注意Session的实际过期时间比设置的过期时间延迟5分钟。 这是必要的，以便在Session过期后可以访问Session的值。 这样做的目的是要确保Session被清理干净，但只有在我们执行任何必要的处理之后再执行清理。

> SessionRepository.getSession(String)方法确保了不会获取过期的Session，这意味着我们在使用一个session时不用检查Sessionde过期时间。

Spring Session依赖于Redis的delete和keyspace notifications，分别触发SessionDeletedEvent和SessionExpiredEvent事件。 SessionDeleEvent或SessionExpiredEvent可以确保与Session关联的资源被清除。 例如，当使用Spring Session与WebSocket集成时，Redis过期或delete事件会导致与Session关联的任何WebSocket连接关闭。

Session本身不直接跟踪过期时间，因为这意味着Session数据将不再可用。 而是使用特殊的Session过期Key。 在我们的例子中，过期Key是：

```
APPEND spring:session:sessions:expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe ""
EXPIRE spring:session:sessions:expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe 1800
```

当Session过期时，Redis的keyspace notification（键空间通知）间通知会通知查找对应的Session，并触发一个SessionDestroyedEvent事件。

依靠Redis的expired有一个问题，就是Redis不保证expired的事件会被触发。具体来说，Redis用于清除过期Session的后台任务是低优先级任务，可能不会触发Session过期。有关其他详细信息，请参阅Redis文档中的expired事件部分。

为了避免 expired事件不能保证被触发的事实，我们可以确保在Session即将过期时访问每个Session 的key。这意味着如果TTL在该key上过期，当我们尝试访问它的key时，Redis将删除该key并触发过期的事件。

因此，每个Session也会记录即将过期的最近时间。这允许后台任务访问即将的过期会话，以确保以更准确的触发Redis过期事件。例如：

```
SADD spring:session:expirations:1439245080000 expires:33fdd1b6-b496-4b33-9f7d-df96679d32fe
EXPIRE spring:session:expirations1439245080000 2100
```

> 在某些情况下，我们没有明确删除key，这是因为可能存在竞争条件导致key被误判为过期。 Redis没有使用分布式锁（这会影响的性能），因此无法保证过期mapping的一致性。 通过简单的访问key的方式，我们确保该key只有在该key的TTL过期时才被删除。

7.6.5 SessionDeletedEvent和SessionExpiredEvent事件

SessionDeletedEvent和SessionExpiredEvent都是SessionDestroyedEvent事件的两种类型。

RedisOperationsSessionRepository支持在Session被删除时触发SessionDeletedEvent，或者在Session过期时触发SessionExpiredEvent事件。 这将确保与Session相关的资源被正确清理。

例如，当与WebSockets集成时，SessionDestroyedEvent负责关闭任何活动的WebSocket连接。

通过Redis Keyspace的事件监听器SessionMessageListener可以触发SessionDeletedEvent或SessionExpiredEvent事件。 为了使其工作，需要启用Generic命令和Expired事件的Redis Keyspace事件。 例如：

```
redis-cli config set notify-keyspace-events Egx
```

如果使用@EnableRedisHttpSession注解，则会自动启动SessionMessageListener监听器并触发必要的Redis Keyspace事件。 但是，在安全的Redis环境中，config命令被禁用。 这意味着Spring Session无法为您配置Redis Keyspace事件。 要禁用自动配置，请将ConfigureRedisAction.NO_OP添加为一个bean。

例如，Java Configuration可以使用以下内容：

```
@Bean
public static ConfigureRedisAction configureRedisAction() {
    return ConfigureRedisAction.NO_OP;
}
```

XML配置如下：

```
<util:constant static-field="org.springframework.session.data.redis.config.ConfigureRedisAction.NO_OP"/>
```

7.6.6 SessionCreatedEvent事件

当Session创建时，将session:channel:created:33fdd1b6-b496-4b33-9f7d-df96679d32fe发送到Redis，其中33fdd1b6-b496-4b33-9f7d-df96679d32fe是Session ID。 事件体就是创建的会话。

如果注册了MessageListener（默认），则RedisOperationsSessionRepository会将Redis消息转换为SessionCreatedEvent。

7.6.7 查看Redis中的Session

安装redis-cli后，可以使用redis-cli查看Redis中的值。 例如，在终端中输入以下内容：

```
$ redis-cli
redis 127.0.0.1:6379> keys *
1) "spring:session:sessions:4fc39ce3-63b3-4e17-b1c4-5e1ed96fb021" ①
2) "spring:session:expirations:1418772300000" ②
```

①该key的后缀是Spring Session的会话标识符。

②此key包含的所有会话ID在1418772300000时被删除。

您还可以查看每个Session的属性。

```
redis 127.0.0.1:6379> hkeys spring:session:sessions:4fc39ce3-63b3-4e17-b1c4-5e1ed96fb021
1) "lastAccessedTime"
2) "creationTime"
3) "maxInactiveInterval"
4) "sessionAttr:username"
redis 127.0.0.1:6379> hget spring:session:sessions:4fc39ce3-63b3-4e17-b1c4-5e1ed96fb021 sessionAttr:username
"\xac\xed\x00\x05t\x00\x03rob"
```

7.7 GemFireOperationsSessionRepository

GemFireOperationsSessionRepository是使用Spring Data的GemFireOperationsSessionRepository实现的SessionRepository。 在Web环境中，这通常与SessionRepositoryFilter结合使用。 该实现通过SessionMessageListener支持SessionDestroyedEvent和SessionCreatedEvent。

7.7.1 GemFire使用索引

关于如何正确定义索引，来提升GemFire的性能超出了本文档的范围，但重要的是要认识到Spring Session Data GemFire可以有效地创建和使用索引来查询和查找会话。

开箱即用，Spring Session GemFire在主体名称上创建1个哈希类型的索引。 查找主体名称的策略有两种不同的表现。 第一个策略是名为FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME的会话属性的值将被索引到相同的索引名称。 例如：

```
String indexName = FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME;
session.setAttribute(indexName, username);
Map<String, ExpiringSession> idToSessions = sessionRepository
        .findByIndexNameAndIndexValue(indexName, username);
```

7.7.2 GemFire & Spring Security使用索引

或者，Spring Session Data GemFire将Spring Security的当前 `Authentication#getName()` 映射到索引FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME。 例如，如果您使用的是Spring Security，可以使用以下方式查找当前用户的Session：

```
SecurityContext context = SecurityContextHolder.getContext();
Authentication authentication = context.getAuthentication();
String indexName = FindByIndexNameSessionRepository.PRINCIPAL_NAME_INDEX_NAME;
Map<String, ExpiringSession> idToSessions = sessionRepository
        .findByIndexNameAndIndexValue(indexName, authentication.getName());
```

7.7.3 GemFire使用自定义索引

这使开发人员能够以编程方式使用GemFireOperationsSessionRepository来查询和查找具有给定主体名称的所有Session。

此外，当开发人员识别出应该由GemFire索引的一个或多个命名的Session属性时，Spring Session Data GemFire将在实现Session的Map类型属性（即任意任意Session属性）上创建一个基于范围的索引。

可以使用@EnableGemFireHttpSession注解中的indexableSessionAttributes属性指定索引的会话属性。 当启用由GemFire支持的Spring Session时，开发人员将此注释添加到其Spring应用程序@Configuration类中。

例如，以下配置：

```
@EnableGemFireHttpSession(indexableSessionAttributes = { "name1", "name2", "name3" })
public class GemFireHttpSessionConfig {
        // ...
}
```

将允许使用以下内容搜索会话：

```
String indexName = "name1";
session.setAttribute(indexName, attrValue);
Map<String, ExpiringSession> idToSessions = sessionRepository
        .findByIndexNameAndIndexValue(indexName, attrValue);
```

> 只有在@EnableGemFireHttpSession注解的indexableSessionAttributes属性中标识的会话属性名称将定义索引。 所有其他会话属性将不被索引。

注意， 存储在可索引的Session属性中的任何值必须实现java.lang.Comparable <T>接口。 如果这些对象值不实现Comparable接口，那么当为具有持久Session数据的区域定义索引时，或者当在运行时尝试将可索引的Session属性分配给不可比较的值时，GemFire将在启动时抛出错误，并且将Session保存到GemFire。

> 没有索引的任何Session属性可以存储非可比较值。

要了解有关GemFire基于范围的索引的更多信息，请参阅在映射字段上创建索引。

要了解有关GemFire索引的更多信息，请参阅使用索引。

7.8 MapSessionRepository

MapSessionRepository允许在Map中持久化ExpiringSession，其中的key是ExpiresSession id，值为ExpiresSession。 该实现可以与ConcurrentHashMap一起用作测试或是一种约定。 或者，它可以与分布式Map实现一起使用。 例如，它可以与Hazelcast一起使用。

7.8.1 初始化MapSessionRepository

创建MapSessionRepository的示例很简单：

```
SessionRepository<? extends ExpiringSession> repository = new MapSessionRepository();
```

7.8.2 Spring Session和 Hazlecast

Hazelcast Sample是一个完整的应用程序，演示如何使用带有Hazelcast的Spring Session。

要运行它，请使用以下命令：

```
./gradlew :samples:hazelcast:tomcatRun
```

Hazelcast Spring Sample是一个完整的应用程序，演示了如何使用Spring Session与Hazelcast和Spring Security集成。

它包括Hazelcast MapListener实现，支持触发SessionCreatedEvent，SessionDeletedEvent和SessionExpiredEvent事件。

要运行它，请使用以下命令：

```
./gradlew :samples:hazelcast-spring:tomcatRun
```

7.9 JdbcOperationsSessionRepository

JdbcOperationsSessionRepository是一个SessionRepository实现，它使用Spring的JdbcOperations在关系数据库中存储Session。 在Web环境中，这通常与SessionRepositoryFilter结合使用。 请注意，此实现不支持发布session事件。

7.9.1 初始化JdbcOperationsSessionRepository实例

如何创建JdbcOperationsSessionRepository实例的典型示例如下所示：

```
JdbcTemplate jdbcTemplate = new JdbcTemplate();

// ... configure JdbcTemplate ...

PlatformTransactionManager transactionManager = new DataSourceTransactionManager();

// ... configure transactionManager ...

SessionRepository<? extends ExpiringSession> repository =
        new JdbcOperationsSessionRepository(jdbcTemplate, transactionManager);
```

有关如何创建和配置JdbcTemplate和PlatformTransactionManager的其他信息，请参阅Spring Framework参考文档。

7.9.2 EnableJdbcHttpSession

在Web环境中，创建JdbcOperationsSessionRepository实例的最简单方法是使用@EnableJdbcHttpSession注解。 Samples and Guides (Start Here)可以找到完整的用法。您可以使用以下属性来自定义配置：

- tableName - Spring Session用于存储会话的数据库表的名称
- maxInactiveIntervalInSeconds - session的过期时间

自定义LobHandler

您可以通过创建一个实现LobHandler接口、名称为springSessionLobHandler的Bean，来实现自定义BLOB处理。

自定义ConversionService

您可以通过提供ConversionService实例来自定义session的序列化和反序列化机制。 在典型的Spring环境中时，默认的ConversionService Bean（名为conversionService）将被自动引用并用于序列化和反序列化。 但是，您可以通过提供一个名为springSessionConversionService的Bean覆盖默认的ConversionService。

7.9.3 存储细节

默认情况下，JdbcOperationsSessionRepository使用SPRING_SESSION和SPRING_SESSION_ATTRIBUTES表来存储会话。 请注意，表名可以按照前面介绍的方式进行定制。 在这种情况下，用于存储属性的表将使用提供的表名命名，后缀为_ATTRIBUTES。 如果需要进一步的自定义，则可以使用set * Query setter方法来定制存储库使用的SQL查询。 在这种情况下，您需要手动配置sessionRepository bean。

由于各种数据库供应商之间的差异，特别是在存储二进制数据时，请确保使用特定于您的数据库的SQL脚本。 大多数主要数据库供应商的脚本打包为org / springframework / session / jdbc / schema - *。sql，其中*是目标数据库类型。

例如，使用PostgreSQL数据库，您将使用以下架构脚本：

```
CREATE TABLE SPRING_SESSION (
        SESSION_ID CHAR(36) NOT NULL,
        CREATION_TIME BIGINT NOT NULL,
        LAST_ACCESS_TIME BIGINT NOT NULL,
        MAX_INACTIVE_INTERVAL INT NOT NULL,
        PRINCIPAL_NAME VARCHAR(100),
        CONSTRAINT SPRING_SESSION_PK PRIMARY KEY (SESSION_ID)
);

CREATE INDEX SPRING_SESSION_IX1 ON SPRING_SESSION (LAST_ACCESS_TIME);

CREATE TABLE SPRING_SESSION_ATTRIBUTES (
        SESSION_ID CHAR(36) NOT NULL,
        ATTRIBUTE_NAME VARCHAR(200) NOT NULL,
        ATTRIBUTE_BYTES BYTEA NOT NULL,
        CONSTRAINT SPRING_SESSION_ATTRIBUTES_PK PRIMARY KEY (SESSION_ID, ATTRIBUTE_NAME),
        CONSTRAINT SPRING_SESSION_ATTRIBUTES_FK FOREIGN KEY (SESSION_ID) REFERENCES SPRING_SESSION(SESSION_ID) ON DELETE CASCADE
);

CREATE INDEX SPRING_SESSION_ATTRIBUTES_IX1 ON SPRING_SESSION_ATTRIBUTES (SESSION_ID);
```

MYSQL数据库如下：

```
CREATE TABLE SPRING_SESSION (
        SESSION_ID CHAR(36) NOT NULL,
        CREATION_TIME BIGINT NOT NULL,
        LAST_ACCESS_TIME BIGINT NOT NULL,
        MAX_INACTIVE_INTERVAL INT NOT NULL,
        PRINCIPAL_NAME VARCHAR(100),
        CONSTRAINT SPRING_SESSION_PK PRIMARY KEY (SESSION_ID)
) ENGINE=InnoDB;

CREATE INDEX SPRING_SESSION_IX1 ON SPRING_SESSION (LAST_ACCESS_TIME);

CREATE TABLE SPRING_SESSION_ATTRIBUTES (
        SESSION_ID CHAR(36) NOT NULL,
        ATTRIBUTE_NAME VARCHAR(200) NOT NULL,
        ATTRIBUTE_BYTES BLOB NOT NULL,
        CONSTRAINT SPRING_SESSION_ATTRIBUTES_PK PRIMARY KEY (SESSION_ID, ATTRIBUTE_NAME),
        CONSTRAINT SPRING_SESSION_ATTRIBUTES_FK FOREIGN KEY (SESSION_ID) REFERENCES SPRING_SESSION(SESSION_ID) ON DELETE CASCADE
) ENGINE=InnoDB;

CREATE INDEX SPRING_SESSION_ATTRIBUTES_IX1 ON SPRING_SESSION_ATTRIBUTES (SESSION_ID);
```

7.9.4 事务管理

JdbcOperationsSessionRepository中的所有JDBC操作都以事务方式执行。 事务的传播为REQUIRES_NEW，以避免对现有事务的干扰（例如，在已经参与只读事务的线程中执行保存操作）。

7.10 HazelcastSessionRepository

8.Spring Session社区

我们欢迎您成为我们社区的一分子。 请在下面找到关于Spring Session的其他信息。

8.1 支持

您可以在StackOverflow上提出问题来获得帮助，提问地址是StackOverflow with the tag spring-session。 同样，我们鼓励通过回答StackOverflow上的问题来帮助别人。

8.2 源码

Spring Session源码的地址是：

https://github.com/spring-projects/spring-session/

8.3 问题跟踪

通过https://github.com/spring-projects/spring-session/issues跟踪github上的问题。

8.4 贡献

我们很感激您Pull Requests。

8.5 License

Spring Session的开源许可证是基于Apache 2.0 license。

8.6 社区扩展

Spring Session OrientDBSpring Session Infinispan

9.最低版本要求

Spring Session的最低要求是：

- Java 5+
- 如果您正在运行Servlet容器（非必需），则Servlet 2.5+
- 如果您使用其他Spring库（非必需），则所需的最低版本为Spring 3.2.14。 当我们针对Spring 3.2.x重新运行所有单元测试时，建议尽可能使用最新的Spring 4.x版本。
- @EnableRedisHttpSession需要Redis 2.8+。 这对于支持Session Expiration是必要的。

> 在核心Spring Session中，只需要对commons-logging的依赖。 有关没有任何其他Spring依赖的Spring Session的示例，请参阅hazelcast示例应用程序。