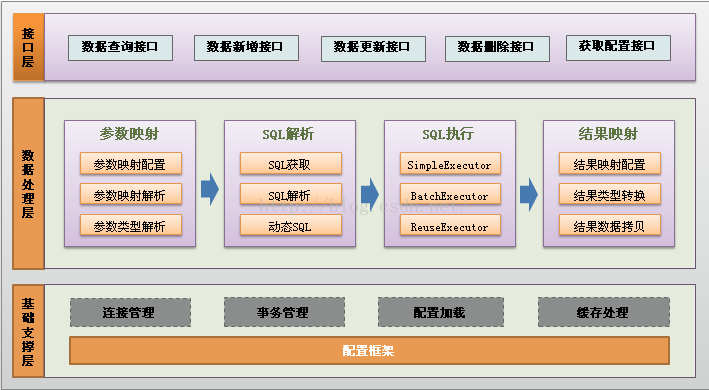
# MyBatis启动流程

## 1.MyBatis简介

<https://blog.csdn.net/CZengze/article/details/52976599>

      MyBatis本是apache的一个开源项目iBatis, 2010年这个项目由apache software foundation 迁移到了google code，并且改名为MyBatis 。它支持普通 SQL查询，存储过程和高级映射的优秀持久层框架。MyBatis 消除了几乎所有的JDBC代码和参数的手工设置以及结果集的检索。MyBatis 使用简单的 XML或注解用于配置和原始映射，将接口和 Java 的POJOs（Plain Old Java Objects，普通的Java对象）映射成数据库中的记录。

## 2.MyBatis功能架构层



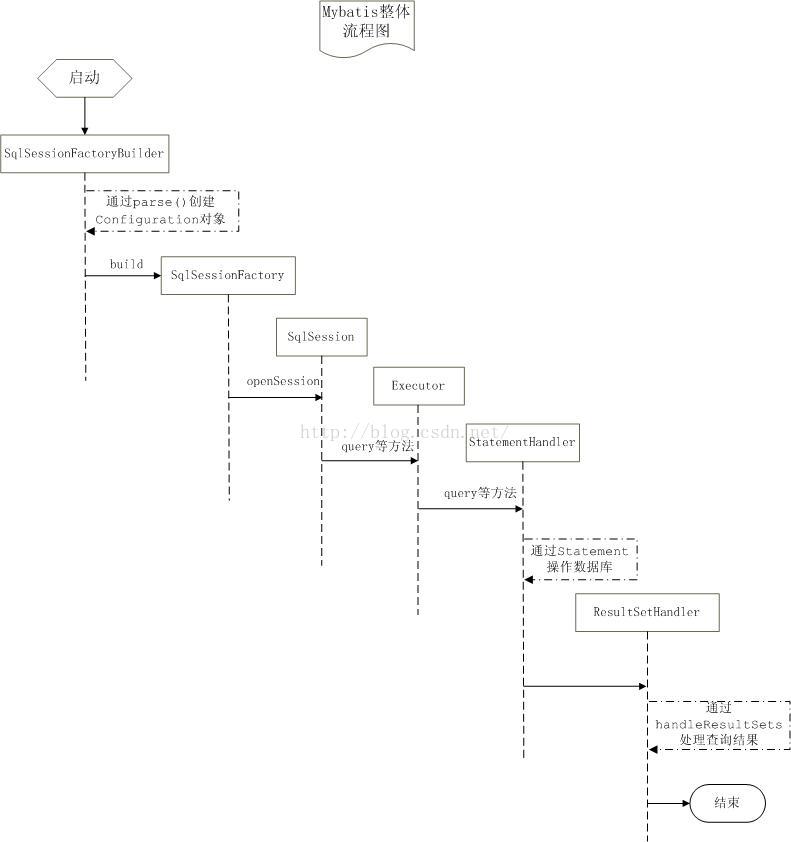
功能架构分为三层：

      1.API接口层：提供给外部使用的接口API，开发人员通过本地API操纵数据库。接口层收到调用请求就会调用数据处理层来完成具体的数据处理。

      2.数据处理层：负责具体的SQL查找，SQL解析，SQL执行和执行结果映射处理。主要是根据调用请求完成一次数据库操作。

      3.基础支撑层：负责基础的功能支撑，包括连接管理，事务管理，配置加载和缓存处理。为上层的数据处理层提供最基础的支撑。

## 3.MyBatis整体流程



## 4.Web启动流程

### web.xml加载过程

      首先，在启动WEB项目的时候，容器(Jetty、Tomcat)首先会读取web.xml项目文件中的配置，当这个步骤没有出错，一个web项目才能启动成功。

1. 启动web项目时，容器会先读取配置文件web.xml中两个节点。

**<context-param></context-param>**和**<listener></listener>**； ；

      2.容器会创建一个ServletContext，使整个项目都能用这个上下文；

      3.容器读到<context-param>转为键值对，交给ServletContext；

      4.容器创建<listener></listener>类实例，即创建监听;

      如：如果想在项目启动就打开数据库，可以在context-param中设置数据库连接方式，在监听类中初始化数据库连接。

      5.在监听的类中会有一个contextInitialized(ServletContextEvent event)初始化方法，在这个方法中可以通过event.getServletContext().getInitParameter("contextConfigLocation") 来得到context-param 设定的值。在这个类中还必须有一个contextDestroyed(ServletContextEvent event) 销毁方法.用于关闭应用前释放资源，比如说数据库连接的关闭；

      6.得到context-param值之后，可以做一些操作。

      context-param>listener>filter>servlet(可以没有context-param，listener必须有)

      <context-param>：含有一对参数名和参数值，用作Servlet上下文初始化参数，参数名在整个web应用中应该是唯一的，在整个web应用的整个周期中，上下文初始化操作都存在，任何Servlet可以随时访问。

### listener简介

      常用web接口事件：

      1.ServletContextListener：监听Web应用的启动和关闭；

      2.ServletContextAttributeListener：监听ServletContext范围内属性的改变；

      3.ServletRequestListener：监听用户的请求；

      4.ServletRequestAttributeListener：监听ServletRequest范围内属性的改变；

      5.HttpSessionListener：监听session的开始和结束；

      6.HttpSessionAttributeListener：监听HttpSession内属性的改变。

      对于整合了MyBatis的Spring里

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

      添加spring监听器，ContextLoaderListener作用是在启动web容器时，自动装配applicationContext.xml里的信息，执行里面的方法。

### filter简介

     用于对用户请求request的预处理，也可以对response后处理，典型的处理链。完整过程是：Filter对用户请求进行预处理，接着将请求HttpServletRequest交给Servlet进行处理并产生响应，最后Filter再对服务器响应HttpServletResponse进行响应。Filter可以负责拦截多个请求和响应，一个请求或响应也可以被多个Filter拦截。

     filter创建分为两步：1.创建Filter处理类；2.web.xml中配置Filter。

     filter配置和Servlet配置类似，但是Servle通常配置一个URL，Filter可以同时配置多个URL。

     在<filter-name>、<filter-class>、<init-param>、<icon>、<display-name>、<description>中

     1.<filter-name>用来定义过滤器的名称，该名称在整个程序中都必须唯一。

      2.<filter-class>元素指定过滤器类的完全限定的名称，即Filter的实现类。

     3.<init-param>为Filter配置参数，与<context-param>具有相同的元素描述符<param-name>和<param-value>。

     4.<filter-mapping>元素用来声明Web应用中的过滤器映射，过滤器被映射到一个servlet或一个URL 模式。这个过滤器的<filter>和<filter-mapping>必须具有相同的<filter-name>，指定该Filter所拦截的URL.

### Servlet简介

      Servlet是个特殊的Java类，继承于HttpServlet，Servlet为了响应客户端请求，一般是GET和POST，必须重写doGet和doPost方法。大部分时候，Servlet对于所有请求响应一样，此时只要重写service()方法就可以响应客户端所有请求。

      创建Servlet有两个时机：1.客户端请求servlet时候，创建servlet实例；2.Web应用启动时，立即创建Servlet实例。

      1.<description>、<display-name>和<icon>

      2.<servlet-name>、<servlet-class>和<jsp-file>元素

      3.<load-on-startup>

     加载Servlet的过程：容器的Context对象对请求路径做出处理，去掉请求URL的上下文路径后，按照路径映射规则和Servlet映射路径做匹配。当匹配成功后，调用这个Servlet处理请求。

## 5.MyBatis启动流程

     现在web.xml里配置context-param里

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>

classpath:applicationContext.xml,

classpath:applicationContext-mybatis.xml

</param-value>

</context-param>

在applicationContext-mybatis.xml里配置，configLocation是用于指定Mybatis的配置文件位置。如果指定了该属性，那么会以该配置文件的内容作为配置信息构建对应的SqlSessionFactoryBuilder，但是后续属性指定的内容会覆盖该配置文件里面指定的对应内容。sqlSessionFactory是把数据源注入给session工厂。

**<bean** id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean"**>**

**<property** name="dataSource" ref="dataSource"**/>**

    <!-- 配置扫描Domain的包路径 -->

**<property** name="typeAliasesPackage" value="com.sankuai.sjst.erp.order.domain"**/>**

    <!-- 配置mybatis配置文件的位置 -->

**<property** name="configLocation" value="classpath:mybatis-config.xml"**/>**

    <!-- 配置扫描Mapper XML的位置 -->

    <!--<property name="mapperLocations"-->

    <!--value="classpath\*:/mapper/\*\*/\*.xml"/>-->

**<property** name="mapperLocations"

              value="classpath:com/sankuai/sjst/erp/order/mapper/\*\*/\*.xml"**/>**

**</bean>**

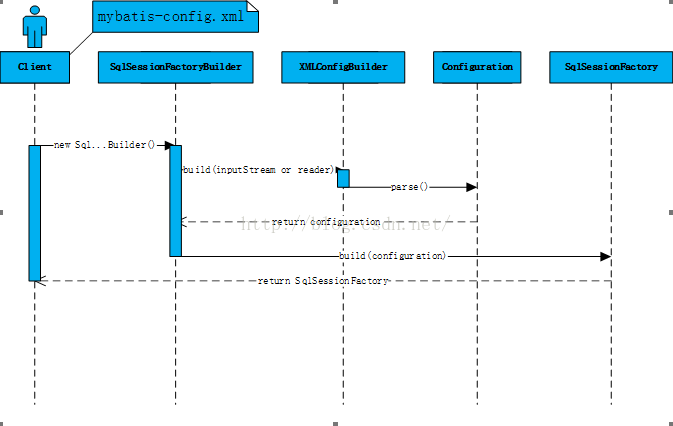
 SqlSessionFactoryBean实现了Spring的FactoryBean接口。这意味着Spring最终返回的不是SqlSessionFactoryBean而是作为factory 的getObject()方法返回的Object。这里相当于代码

SqlSessionFactoryBean factoryBean = new SqlSessionFactoryBean();

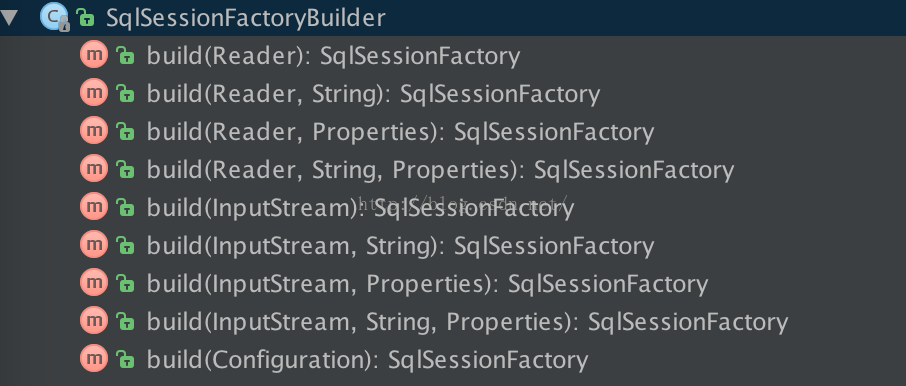
SqlSessionFactory sessionFactory = factoryBean.getObject();

SqlSessionFactory 一旦被创建就应该在应用的运行期间一直存在，没有任何理由对它进行清除或重建。使用 SqlSessionFactory 的最佳实践是在应用运行期间不要重复创建多次，多次重建 SqlSessionFactory 被视为一种代码“坏味道（bad smell）”。因此 SqlSessionFactory 的最佳范围是应用范围。有很多方法可以做到，最简单的就是使用单例模式或者静态单例模式。

      MyBatis初始化流程图



使用Mybatis实例，第一步就是要产生SqlSessionFactory类的实例，通过调用SqlSessionFactoryBuilder的builder方法来完成。



SqlSessionFactoryBuilder类负责构建SqlSessionFactory,并且提供了多个build的重载方法。根据缺省去重后，有三类比较有效。

public SqlSessionFactory build(Reader reader, String environment, Properties properties) {

try {

XMLConfigBuilder parser = new XMLConfigBuilder(reader, environment, properties);

//configuration是parser.parse()

return build(parser.parse());

} catch (Exception e) {

throw ExceptionFactory.wrapException("Error building SqlSession.", e);

} finally {

ErrorContext.instance().reset();

try {

reader.close();

} catch (IOException e) {

// Intentionally ignore. Prefer previous error.

}

}

}

public SqlSessionFactory build(InputStream inputStream, String environment, Properties properties) {

try {

XMLConfigBuilder parser = new XMLConfigBuilder(inputStream, environment, properties);

return build(parser.parse());

} catch (Exception e) {

throw ExceptionFactory.wrapException("Error building SqlSession.", e);

} finally {

ErrorContext.instance().reset();

try {

inputStream.close();

} catch (IOException e) {

// Intentionally ignore. Prefer previous error.

}

}

}

public SqlSessionFactory build(Configuration config) {

return new DefaultSqlSessionFactory(config);

｝

配置信息给三种形式提供sqlSessionFactory的build方法，分别是InputStream(字节流）、Reader（字符流）、Configuration（类）。字节流和字符流都为读取配置文件的方式。

 从配置文件中可以想到的启动方法是：

String resource = "org/mybatis/example/mybatis-config.xml";

InputStream inputStream = Resources.getResourceAsStream(resource);

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream) ;

XML文件的构造方式，通过从XML中读取信息的工作之后，也是构造出Configuration对象之后再继续进行SqlSessionFactory的构建工作的。

 编程Configuration方式：

DataSource dataSource = BlogDataSourceFactory.getBlogDataSource();

TransactionFactory transactionFactory = new JdbcTransactionFactory();

Environment environment = new Environment("development", transactionFactory, dataSource);

Configuration configuration = new Configuration(environment);

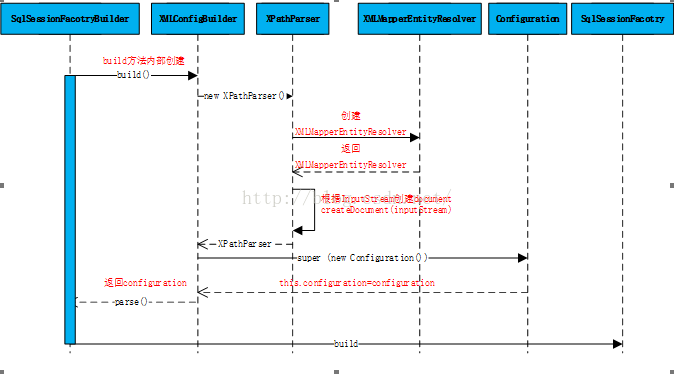
configuration.addMapper(BlogMapper.class);

SqlSessionFactory sqlSessionFactory = new SqlSessionFactoryBuilder().build(configuration);

SqlSessionFactory在mybatis的默认实现类为

org.apache.ibatis.session.defaults.DefaultSqlSessionFactory , 其构造过程主要是注入了Configuration的实例对象，Configuration的实例对象即可通过解析xml配置文件产生，也可能通过代码直接构造。

      解析流程图：



即流程步骤为：

      1.创建MybatisDTD文件实体类：XMLMapperEntityResolver。

      2.根据配置文件流信息和上一步创建的EntityResolver创建配置文件解析类：XPathParser用于解析配置文件内容。

      3.将前两部创建的对象作为XMLConfigBuilder的构造函数参数传递、创建XMLConfigBuiler对象。

      4.调用XMLConfigBuilder.parse()创建Configuration对象并将配置文件信息装配到Configuration对象中。

      对于InputStream类别的XMLConfigBuilder的构造方法是：

public XMLConfigBuilder(InputStream inputStream, String environment, Properties props) {

this(new XPathParser(inputStream, true, props, new XMLMapperEntityResolver()), environment, props);

｝

这里可以看出要构建 XMLConfigBuilder，需要先创建XMLMapperEntityResolver。

      XMLMapperEntityResolver的创建：Mybatis的DTD文件加载到一个私有集合中private static final Map<String, String> doctypeMap = new HashMap<String, String>();并向外提供一个用户获取DTD的InputSource的方法public InputSource resolveEntity(String publicId, String systemId)

private static final Map<String, String> doctypeMap = new HashMap<String, String>();

//

//

public InputSource resolveEntity(String publicId, String systemId) throws SAXException {

if (publicId != null) publicId = publicId.toUpperCase(Locale.ENGLISH);

if (systemId != null) systemId = systemId.toUpperCase(Locale.ENGLISH);

InputSource source = null;

try {

String path = doctypeMap.get(publicId);

source = getInputSource(path, source);

if (source == null) {

path = doctypeMap.get(systemId);

source = getInputSource(path, source);

}

} catch (Exception e) {

throw new SAXException(e.toString());

}

return source;

｝

XPathParser的创建，EntityResolver就是前面的XMLMapperEntityResolver，InputStream是配置文件流信息。

public XPathParser(Reader reader, boolean validation, Properties variables, EntityResolver entityResolver) {

commonConstructor(validation, variables, entityResolver);

this.document = createDocument(new InputSource(reader));

｝

 commonConstructor的信息：

private void commonConstructor(boolean validation, Properties variables, EntityResolver entityResolver) {

this.validation = validation;

this.entityResolver = entityResolver;

this.variables = variables;

XPathFactory factory = XPathFactory.newInstance();

this.xpath = factory.newXPath();

｝

根据InputSource创建Document：

private Document createDocument(InputSource inputSource) {

// important: this must only be called AFTER common constructor

try {

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();

factory.setValidating(validation);

factory.setNamespaceAware(false);

factory.setIgnoringComments(true);

factory.setIgnoringElementContentWhitespace(false);

factory.setCoalescing(false);

factory.setExpandEntityReferences(true);

DocumentBuilder builder = factory.newDocumentBuilder();

builder.setEntityResolver(entityResolver);

builder.setErrorHandler(new ErrorHandler() {

public void error(SAXParseException exception) throws SAXException {

throw exception;

}

public void fatalError(SAXParseException exception) throws SAXException {

throw exception;

}

public void warning(SAXParseException exception) throws SAXException {

}

});

return builder.parse(inputSource);

} catch (Exception e) {

throw new BuilderException("Error creating document instance. Cause: " + e, e);

}

｝

当XPathParser创建完成后，回到真正执行XMLConfigBuilder创建的方法：

private XMLConfigBuilder(XPathParser parser, String environment, Properties props) {

super(new Configuration());

ErrorContext.instance().resource("SQL Mapper Configuration");

this.configuration.setVariables(props);

this.parsed = false;

this.environment = environment;

this.parser = parser;

｝

调用了Configuration无参构造方法创建其实例对象，设置XMLConfigBuilder解析装配Configuration需要用的属性，其中this.parser = parser;是前面实例化好的XPathParser。

      当XMLConfigBuilder实例化好了之后，就是解析配置文件，装配Configuration。

public Configuration parse() {

if (parsed) {

throw new BuilderException("Each XMLConfigBuilder can only be used once.");

}

parsed = true;

parseConfiguration(parser.evalNode("/configuration"));

return configuration;

}

      parseConfiguration是获取配置文件中configuration节点所有信息包括其子节点，将MyBatis配置文件各个配置项解析并装配到Configuration中。

private void parseConfiguration(XNode root) {

try {

propertiesElement(root.evalNode("properties")); //issue #117 read properties first

typeAliasesElement(root.evalNode("typeAliases"));

pluginElement(root.evalNode("plugins"));

objectFactoryElement(root.evalNode("objectFactory"));

objectWrapperFactoryElement(root.evalNode("objectWrapperFactory"));

settingsElement(root.evalNode("settings"));

environmentsElement(root.evalNode("environments")); // read it after objectFactory and objectWrapperFactory issue #631

databaseIdProviderElement(root.evalNode("databaseIdProvider"));

typeHandlerElement(root.evalNode("typeHandlers"));

mapperElement(root.evalNode("mappers"));

} catch (Exception e) {

throw new BuilderException("Error parsing SQL Mapper Configuration. Cause: " + e, e);

}

}

       settingElements，是其中的一个例子，将MyBatis配置文件中<settings></settings>解析到Configuration中。

private void settingsElement(XNode context) throws Exception {

if (context != null) {

Properties props = context.getChildrenAsProperties();

// Check that all settings are known to the configuration class

MetaClass metaConfig = MetaClass.forClass(Configuration.class);

for (Object key : props.keySet()) {

if (!metaConfig.hasSetter(String.valueOf(key))) {

throw new BuilderException("The setting " + key + " is not known. Make sure you spelled it correctly (case sensitive).");

}

}

configuration.setAutoMappingBehavior(AutoMappingBehavior.valueOf(props.getProperty("autoMappingBehavior", "PARTIAL")));

configuration.setCacheEnabled(booleanValueOf(props.getProperty("cacheEnabled"), true));

configuration.setProxyFactory((ProxyFactory) createInstance(props.getProperty("proxyFactory")));

configuration.setLazyLoadingEnabled(booleanValueOf(props.getProperty("lazyLoadingEnabled"), false));

configuration.setAggressiveLazyLoading(booleanValueOf(props.getProperty("aggressiveLazyLoading"), true));

configuration.setMultipleResultSetsEnabled(booleanValueOf(props.getProperty("multipleResultSetsEnabled"), true));

configuration.setUseColumnLabel(booleanValueOf(props.getProperty("useColumnLabel"), true));

configuration.setUseGeneratedKeys(booleanValueOf(props.getProperty("useGeneratedKeys"), false));

configuration.setDefaultExecutorType(ExecutorType.valueOf(props.getProperty("defaultExecutorType", "SIMPLE")));

configuration.setDefaultStatementTimeout(integerValueOf(props.getProperty("defaultStatementTimeout"), null));

configuration.setMapUnderscoreToCamelCase(booleanValueOf(props.getProperty("mapUnderscoreToCamelCase"), false));

configuration.setSafeRowBoundsEnabled(booleanValueOf(props.getProperty("safeRowBoundsEnabled"), false));

configuration.setLocalCacheScope(LocalCacheScope.valueOf(props.getProperty("localCacheScope", "SESSION")));

configuration.setJdbcTypeForNull(JdbcType.valueOf(props.getProperty("jdbcTypeForNull", "OTHER")));

configuration.setLazyLoadTriggerMethods(stringSetValueOf(props.getProperty("lazyLoadTriggerMethods"), "equals,clone,hashCode,toString"));

configuration.setSafeResultHandlerEnabled(booleanValueOf(props.getProperty("safeResultHandlerEnabled"), true));

configuration.setDefaultScriptingLanguage(resolveClass(props.getProperty("defaultScriptingLanguage")));

configuration.setCallSettersOnNulls(booleanValueOf(props.getProperty("callSettersOnNulls"), false));

configuration.setLogPrefix(props.getProperty("logPrefix"));

configuration.setLogImpl(resolveClass(props.getProperty("logImpl")));

configuration.setConfigurationFactory(resolveClass(props.getProperty("configurationFactory")));

}

}

       配置装配完成后，返回Configuration

public SqlSessionFactory build(Configuration config) {

return new DefaultSqlSessionFactory(config);

}

      返回最终生成的DefaultSqlSessionFactory。

      MyBatis启动流程，里面使用了大量的方法重载，可以根据不同的场景来生成SqlSessionFactory。