关于spring

1.关于spring框架?

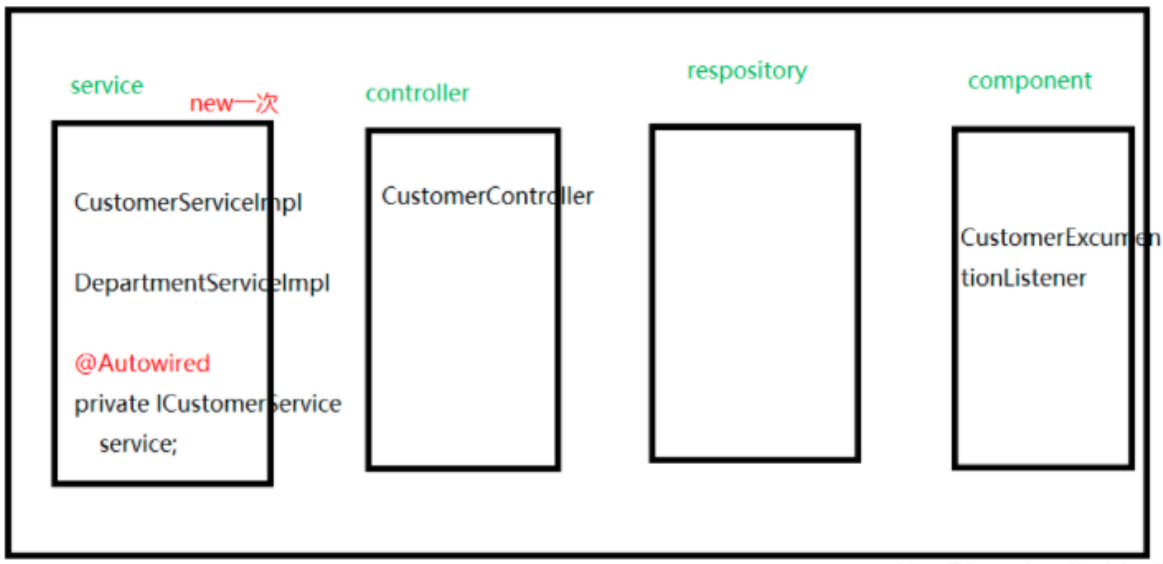
目的：解决企业应用开发的复杂性

功能：使用基本的JavaBean代替EJB（基于分布式事务处理的企业级应用程序的组件），并提供了更多的企业应用功能

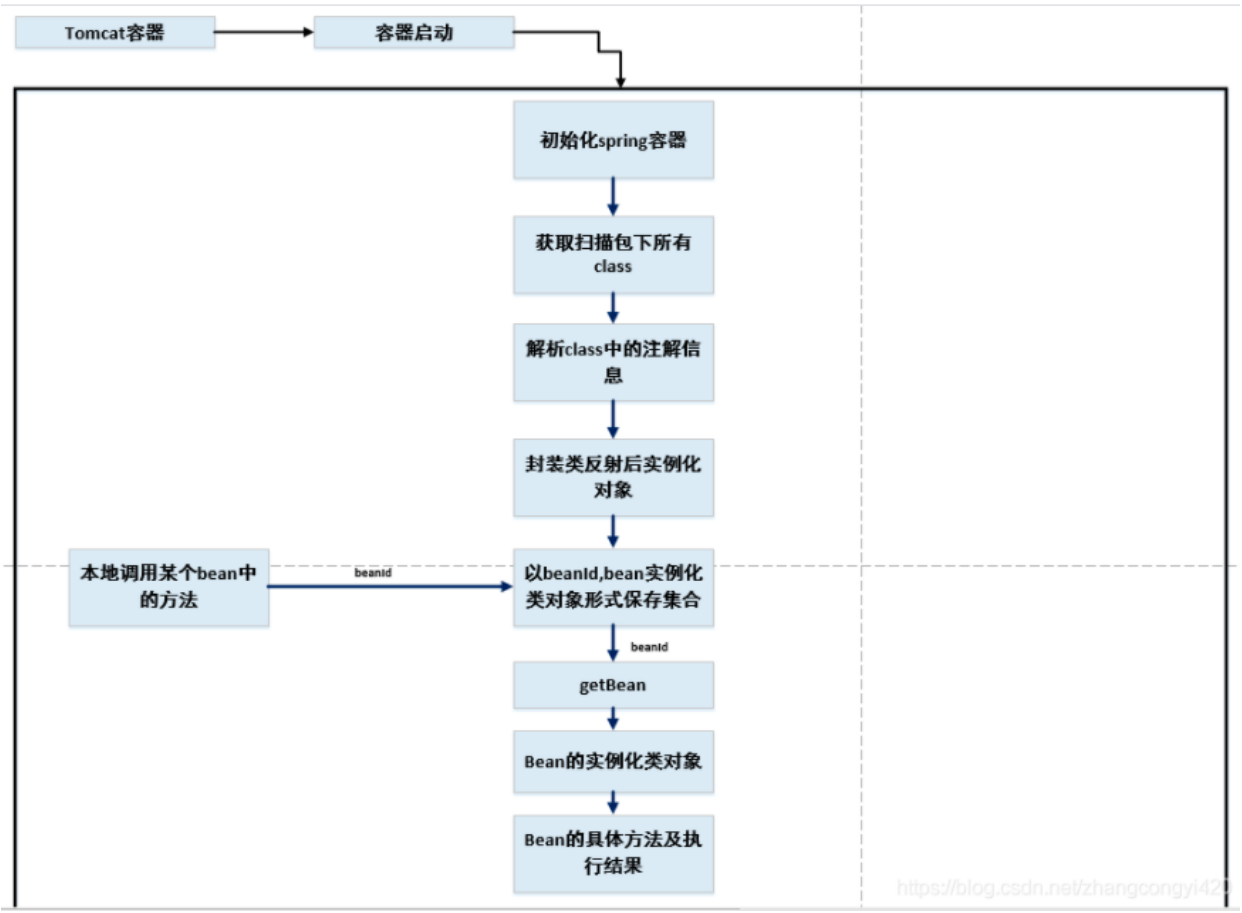
范围：任何Java应用

2.spring框架到底做了些什么?

> **关于Ioc**
>
> **spring ioc指的是控制反转，IOC容器负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。交由Spring容器统一进行管理，从而实现松耦合**
>
> 明确的点：**“控制反转”，不是什么技术，而是一种设计思想。**
>
> 搞明白的点：**“谁控制谁，控制什么，为何是反转（有反转就应该有正转了），哪些方面反转了”**
>
> ●谁控制谁，控制什么：传统Java SE程序设计，我们直接在对象内部通过new进行创建对象，是程序主动去创建依赖对象；而IoC是有专门一个容器来创建这些对象，即由Ioc容器来控制对 象的创建；
>
> 谁控制谁？当然是IoC 容器控制了对象；
>
> 控制什么？那就是主要控制了外部资源获取（不只是对象包括比如文件等）。
>
> 其**底层原理**是Spring有一个容器为IOC，这个容器中开辟了很多个很重要的注解，其主要的为四大注解
>
> 分别为 @Service （注入dao）
>
> 用于标注服务层，主要用来进行业务的逻辑处理
>
> @Controller 控制器（注入服务）
>
> 用于标注控制层，相当于struts中的action层
>
> @Respository（实现dao访问）
>
> 用于标注数据访问层，也可以说用于标注数据访问组件，即DAO组件
>
> @Component（把普通pojo实例化到spring容器中，相当于配置文件中的`）
>
> 泛指各种组件，就是说当我们的类不属于各种归类的时候（不属于@Controller、@Services等的时候），我们就可以使用@Component来标注这个类。



了解了以上部分我们看一下实现原理：

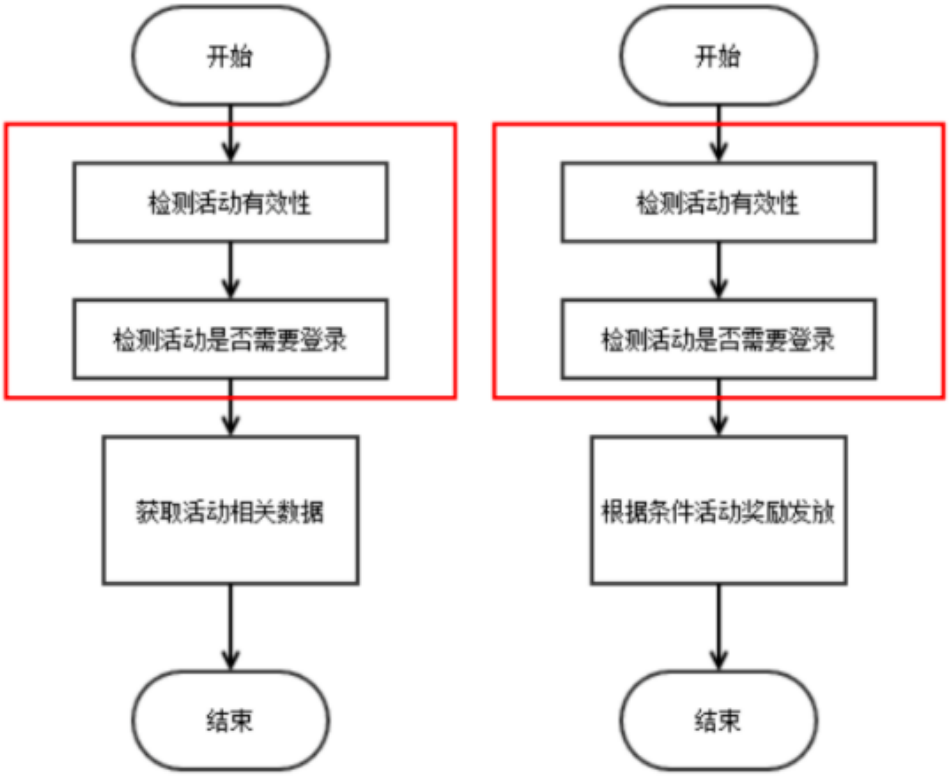


●为何是反转，哪些方面反转了：有反转就有正转，传统应用程序是由我们自己在对象中主动控制去直接获取依赖对象，也就是正转；而反转则是由容器来帮忙创建及注入依赖对象；

为何是反转？因为由容器帮我们查找及注入依赖对象，对象只是被动的接受依赖对象，所以是反转；

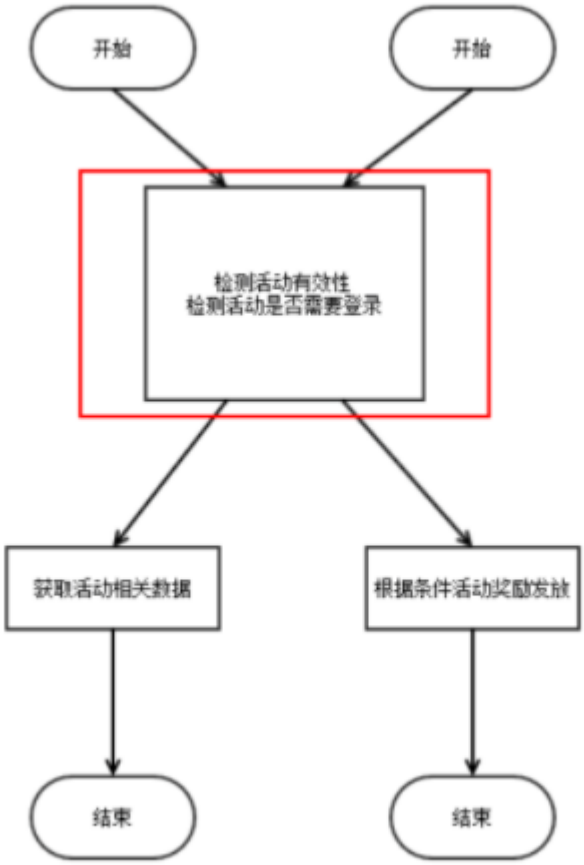
哪些方面反转了？依赖对象的获取被反转了。

关于aop



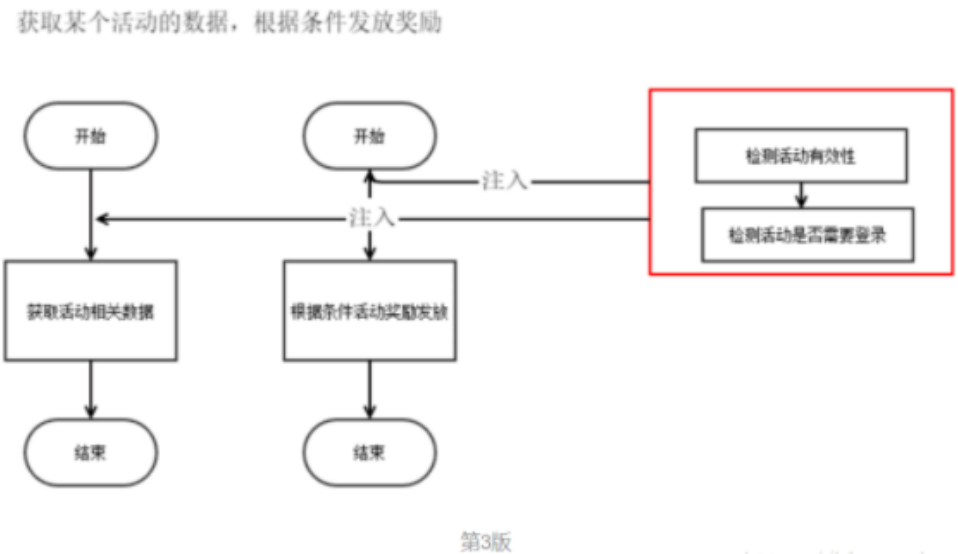
第1版

我们可以调用接口



第2版

同样有个问题，我虽然不用每次都copy代码了，但是，每个接口总得要调用这个方法吧。于是就有了切面的概念，我将方法注入到接口调用的某个地方（切点）



第3版

关于aop的名词解释：

这里还是先给出一个比较专业的概念定义：

- `Aspect`（切面）： Aspect 声明类似于 Java 中的类声明，在 Aspect 中会包含着一些 Pointcut 以及相应的 Advice。

- `Joint point`（连接点）：表示在程序中明确定义的点，典型的包括方法调用，对类成员的访问以及异常处理程序块的执行等等，它自身还可以嵌套其它 joint point。
- `Pointcut`（切点）：表示一组 joint point，这些 joint point 或是通过逻辑关系组合起来，或是通过通配、正则表达式等方式集中起来，它定义了相应的 Advice 将要发生的地方。
- `Advice`（增强）：Advice 定义了在 `Pointcut` 里面定义的程序点具体要做的操作，它通过 before、after 和 around 来区别是在每个 joint point 之前、之后还是代替执行的代码。
- `Target`（目标对象）：织入 `Advice` 的目标对象..。
- `Weaving`（织入）：将 `Aspect` 和其他对象连接起来, 并创建 `Advice`d object 的过程

然后举一个容易理解的例子： 看完了上面的理论部分知识, 我相信还是会有不少朋友感觉到 AOP 的概念还是很模糊, 对 AOP 中的各种概念理解的还不是很透彻. 其实这很正常, 因为 AOP 中的概念是在是太多了, 我当时也是花了老大劲才梳理清楚的. 下面我以一个简单的例子来比喻一下 AOP 中 `Aspect`, `Joint point`, `Pointcut` 与 `Advice`之间的关系. 让我们来假设一下, 从前有一个叫爪哇的小县城, 在一个月黑风高的晚上, 这个县城中发生了命案. 作案的凶手十分狡猾, 现场没有留下什么有价值的线索. 不过万幸的是, 刚从隔壁回来的老王恰好在这时候无意中发现了凶手行凶的过程, 但是由于天色已晚, 加上凶手蒙着面, 老王并没有看清凶手的面目, 只知道凶手是个男性, 身高约七尺五寸. 爪哇县的县令根据老王的描述, 对守门的士兵下命令说: 凡是发现有身高七尺五寸的男性, 都要抓过来审问. 士兵当然不敢违背县令的命令, 只好把进出城的所有符合条件的人都抓了起来.

来让我们看一下上面的一个小故事和 AOP 到底有什么对应关系. 首先我们知道, 在 Spring AOP 中 `Joint point` 指代的是所有方法的执行点, 而 point cut 是一个描述信息, 它修饰的是 `Joint point`, 通过 point cut, 我们就可以确定哪些 `Joint point` 可以被织入 `Advice`. 对应到我们在上面举的例子, 我们可以做一个简单的类比, **Joint point 就相当于 爪哇的小县城里的百姓**,**pointcut 就相当于 老王所做的指控, 即凶手是个男性, 身高约七尺五寸**, **而 Advice 则是施加在符合老王所描述的嫌疑人的动作: 抓过来审问**. 为什么可以这样类比呢?

- `Joint point`： 爪哇的小县城里的百姓: 因为根据定义, `Joint point` 是所有可能被织入 `Advice` 的候选的点, 在 Spring AOP中, 则可以认为所有方法执行点都是 `Joint point`. 而在我们上面的例子中, 命案发生在小县城中, 按理说在此县城中的所有人都有可能是嫌疑人.
- `Pointcut`： 男性, 身高约七尺五寸: 我们知道, 所有的方法(joint point) 都可以织入 `Advice`, 但是我们并不希望在所有方法上都织入 `Advice`, 而 `Pointcut` 的作用就是提供一组规则来匹配joinpoint, 给满足规则的 joinpoint 添加 `Advice`. 同理, 对于县令来说, 他再昏庸, 也知道不能把县城中的所有百姓都抓起来审问, 而是根据凶手是个男性, 身高约七尺五寸, 把符合条件的人抓起来. 在这里 凶手是个男性, 身高约七尺五寸 就是一个修饰谓语, 它限定了凶手的范围, 满足此修饰规则的百姓都是嫌疑人, 都需要抓起来审问.
- `Advice`： 抓过来审问, `Advice` 是一个动作, 即一段 Java 代码, 这段 Java 代码是作用于 point cut 所限定的那些 `Joint point` 上的. 同理, 对比到我们的例子中, 抓过来审问 这个动作就是对作用于那些满足男性, 身高约七尺五寸 的爪哇的小县城里的百姓.

  Advice 的类型

  - `before advice`, 在 join point 前被执行的 advice. 虽然 before advice 是在 join point 前被执行, 但是它并不能够阻止 join point 的执行, 除非发生了异常(即我们在 before advice 代码中, 不能人为地决定是否继续执行 join point 中的代码)
  - `after return advice`, 在一个 join point 正常返回后执行的 advice
  - `after throwing advice`, 当一个 join point 抛出异常后执行的 advice
  - `after(final) advice`, 无论一个 join point 是正常退出还是发生了异常, 都会被执行的 advice.
  - `around advice`, 在 join point 前和 joint point 退出后都执行的 advice. 这个是最常用的 advice.
  - `introduction`， introduction可以为原有的对象增加新的属性和方法。

- `Aspect`:: `Aspect` 是 point cut 与 `Advice` 的组合, 因此在这里我们就可以类比: “根据老王的线索, 凡是发现有身高七尺五寸的男性, 都要抓过来审问” 这一整个动作可以被认为是一个 `Aspect`.

3.关于spring的事务?

1.框架中对事务处理【本质】

封装这种复杂、复用性比较低的代码

2.spring事务 两大配置用法

（1）【声明式事务处理】事务注解、xml配置

```
@Transactional
public int insert(Student student) {
    String sql="insert into student(studentid,studentno,stuname) values(?,?,?)";
    int update = getJdbcTemplate().update(sql, student.getStudentId(), student.getStudentNo(), student.getStuName());
    //System.out.println(7/0);
    return update;
}
```

（2）【编程式事务管理】java代码

```
    @Autowired
    DataSourceTransactionManager dataSourceTransactionManager;
    @Autowired
    TransactionDefinition transactionDefinition;
//    @Transactional
    public int insert(Student student) {
        TransactionStatus transaction = dataSourceTransactionManager.getTransaction(transactionDefinition);
        int update = 0;
        try {
            String sql="insert into student(studentid,studentno,stuname) values(?,?,?)";
            update = getJdbcTemplate().update(sql, student.getStudentId(), student.getStudentNo(), student.getStuName());
            int i = 1/0;
            System.out.println("提交事务");
            dataSourceTransactionManager.commit(transaction);
            return update;
        } catch (Exception e) {
            e.getMessage();
            System.out.println("回滚事务");
            dataSourceTransactionManager.rollback(transaction);
        }

        //System.out.println(7/0);
        return update;
    }
```

4.Spring的优缺点是什么?

优点

方便解耦，简化开发

Spring就是一个大工厂，可以将所有对象的创建和依赖关系的维护，交给Spring管理。

AOP编程的支持

Spring提供面向切面编程，可以方便的实现对程序进行权限拦截、运行监控等功能。

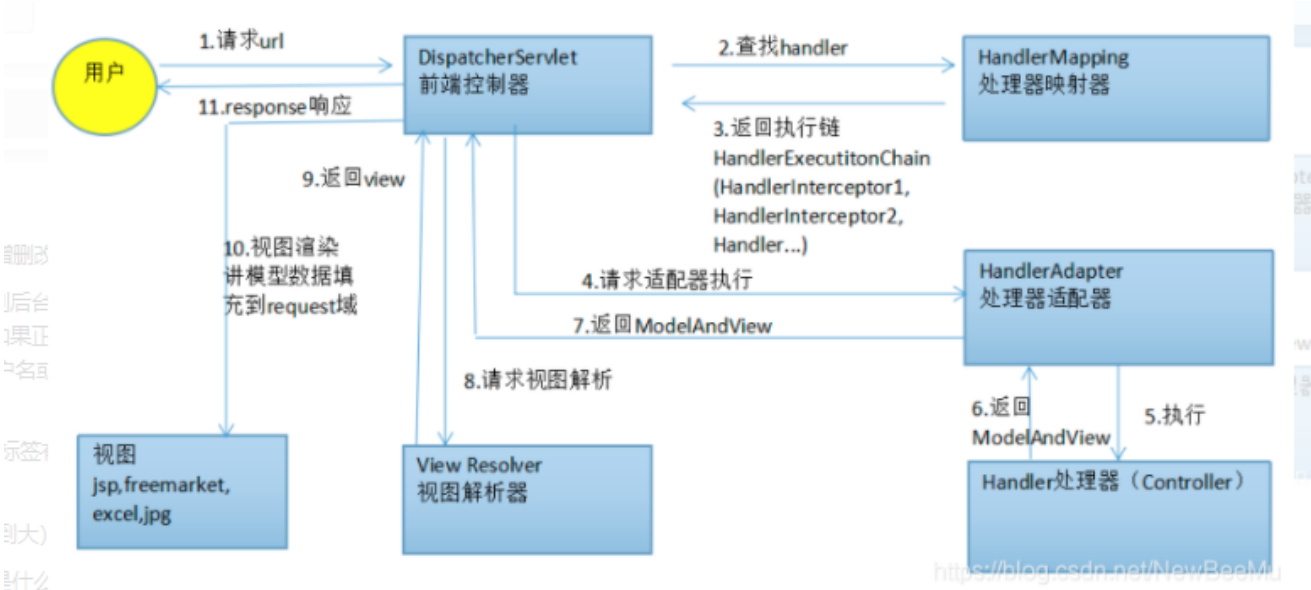
> 声明式事务的支持
>
> 只需要通过配置就可以完成对事务的管理，而无需手动编程。
>
> 方便程序的测试
>
> Spring对Junit4支持，可以通过注解方便的测试Spring程序。
>
> 方便集成各种优秀框架
>
> Spring不排斥各种优秀的开源框架，其内部提供了对各种优秀框架的直接支持（如：Struts、Hibernate、MyBatis等）。
>
> 降低JavaEE API的使用难度
>
> Spring对JavaEE开发中非常难用的一些API（JDBC、JavaMail、远程调用等），都提供了封装，使这些API应用难度大大降低。
>
> 缺点
>
> Spring明明一个很轻量级的框架，却给人感觉大而全
>
> Spring依赖反射，反射影响性能
>
> 使用门槛升高，入门Spring需要较长时间

5.Spring 框架中都用到了哪些设计模式?

> 工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；
>
> 单例模式：Bean默认为单例模式。
>
> 代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；
>
> 模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。
>
> 观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如Spring中listener的实现–ApplicationListener。

关于springmvc

1.**SpringMVC执行流程框图：**

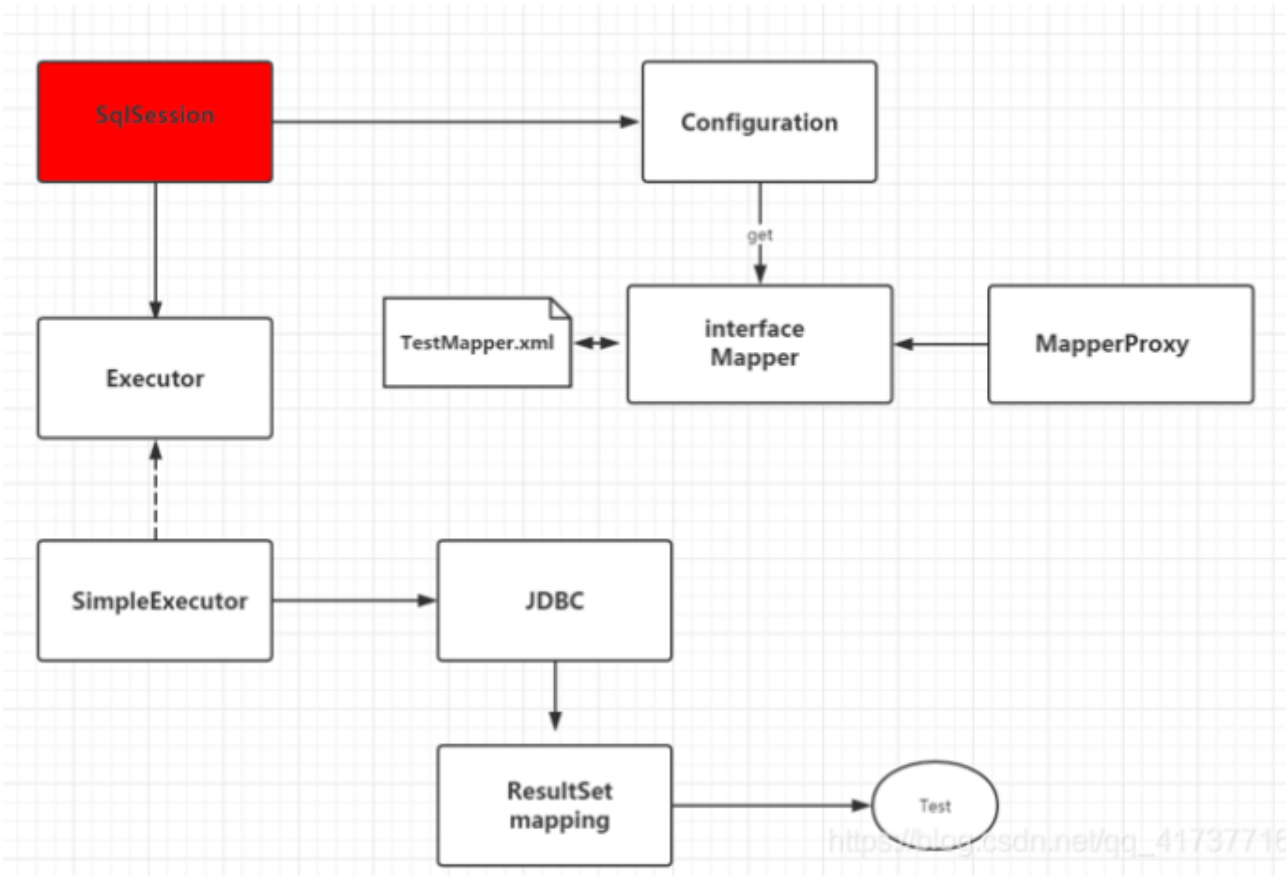


> **SpringMVC流程：** 01、用户发送出请求到前端控制器DispatcherServlet。（不需要程序员开发，在web.xml中配置。作用：接收请求，响应结果，相当于转发器，中央处理器。） 02、DispatcherServlet收到请求调用HandlerMapping（处理器映射器）。(不需要程序员开发,由框架提供。 作用：根据请求的url查找Handler(处理器/Controller)，可以通过XML和注解方式来映射。） 03、HandlerMapping找到具体的控制器(可查找xml配置或注解配置)，生成处理器对象的执行链(如果有)，再一起返回给DispatcherServlet。 04、DispatcherServlet调用HandlerAdapter（处理器适配器）。(不需要程序员开发,由框架提供。 作用：按照特定规则（HandlerAdapter要求的规则）去执行Controller。） 05、HandlerAdapter经过适配调用具体的处理器（controller）。(需要工程师开发 注意：编写Handler时按照HandlerAdapter的要求去做，这样适配器才可以去正确执行Handler。 作用：接受用户请求信息，调用业务方法处理请求，也称之为后端控制器。） 06、Controller执行完成返回ModelAndView对象。 07、HandlerAdapter将Controller执行结果ModelAndView返回给DispatcherServlet。 08、DispatcherServlet将ModelAndView传给ViewReslover（视图解析器）。(不需要程序员开发,由框架提供 作用：进行视图解析，把逻辑视图名解析成真正的物理视图。 SpringMVC框架支持多种View视图技术，包括：jstlView、freemarkerView、pdfView等。） 09、ViewReslover解析后返回具体View（视图）。(需要工程师开发 作用：把数据展现给用户的页面View是一个接口，实现类支持不同的View技术（jsp、freemarker、pdf等）） 10、DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。 11、DispatcherServlet响应用户。

关于mybatis

1.mybatis工作流程

SqlSession为主要的调配者，持有Configuration与Executor，先是创建Mapper委托Configuration去以MapperProxy给Mapper接口做动态代理，底层查询方法根据mapper.xml的查询类型执行SqlSession的查询方法，而SqlSession在查询时又委托Executor去做实际的查询，Executor会使用Statement查询结果集，然后使用ResultSetmapping做结果集的映射POJO，然后返回给SqlSession，因为动态代理，所以mapper的方法实际是SqlSession执行的查询方法，所以这时候SqlSession返回给方法查询结果，表面看起来像是Mapper的方法返回的结果，实际上却是SqlSession在做事情。



2.mybatis中"#（）"和"$（）"的区别?

- #{}： 根据参数的类型进行处理，比如传入String类型，则会为参数加上双引号。#{} 传参在进行SQL预编译时，会把参数部分用一个占位符？代替，这样可以防止 SQL注入。

  `示例1：` `执行SQL：Select * from emp where name = #{employeeName}` `参数：employeeName=>Smith` `解析后执行的SQL：Select * from emp where name = ?` `执行SQL：Select * from emp where name = ${employeeName}` `参数：employeeName传入值为：Smith` `解析后执行的SQL：Select * from emp where name =Smith`

  `综上所述、${}方式会引发SQL注入的问题、同时也会影响SQL语句的预编译，所以从安全性和性能的角度出发，能使用#{}的情况下就不要使用${}`

- ${}： 将参数取出不做任何处理，直接放入语句中，就是简单的字符串替换，并且该参数会参加SQL的预编译，需要手动过滤参数防止 SQL注入。
- 因此 mybatis 中优先使用 #{}；当需要动态传入 表名或列名时，再考虑使用 ${} 。

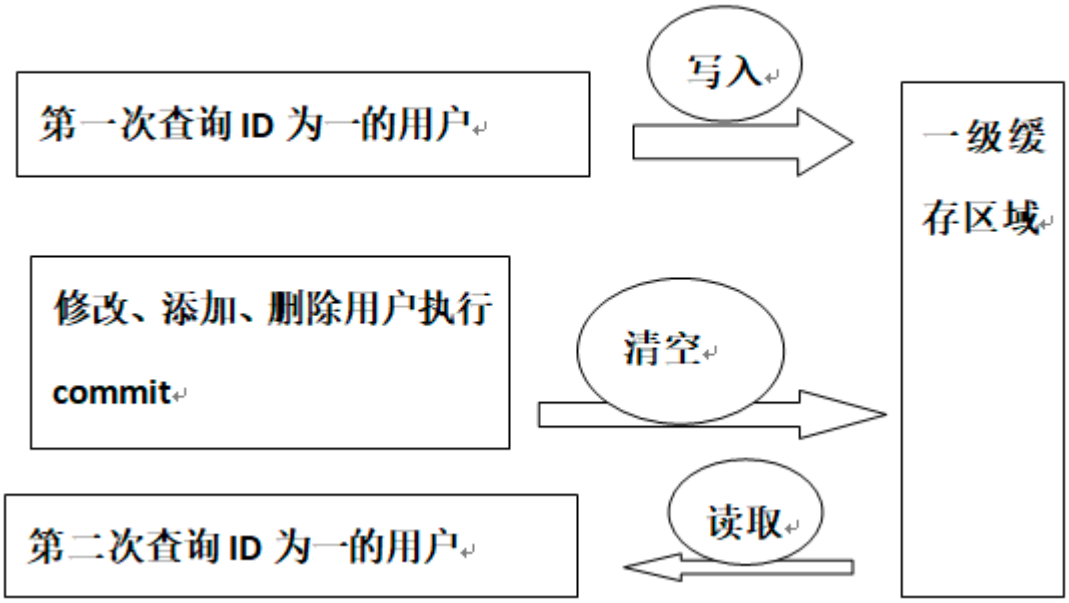
  比如，动态SQL中的字段名，如：ORDER BY ${columnName}

3.mybatis的一级缓存和二级缓存?

先说缓存，合理使用缓存是优化中最常见的，将从数据库中查询出来的数据放入缓存中，下次使用时不必从数据库查询，而是直接从缓存中读取，避免频繁操作数据库，减轻数据库的压力，同时提高系统性能。

一级缓存：是SqlSession级别的缓存。在操作数据库时需要构造sqlSession对象，在对象中有一个数据结构用于存储缓存数据。不同的sqlSession之间的缓存数据区域是互相不影响的。也就是他只能作用在同一个sqlSession中，不同的sqlSession中的缓存是互相不能读取的。

一级缓存的工作原理：



用户发起查询请求，查找某条数据，sqlSession先去缓存中查找，是否有该数据，如果有，读取；

如果没有，从数据库中查询，并将查询到的数据放入一级缓存区域，供下次查找使用。

但sqlSession执行commit，即增删改操作时会清空缓存。这么做的目的是避免脏读。

如果commit不清空缓存，会有以下场景：A查询了某商品库存为10件，并将10件库存的数据存入缓存中，之后被客户买走了10件，数据被delete了，但是下次查询这件商品时，并不从数据库中查询，而是从缓存中查询，就会出现错误。

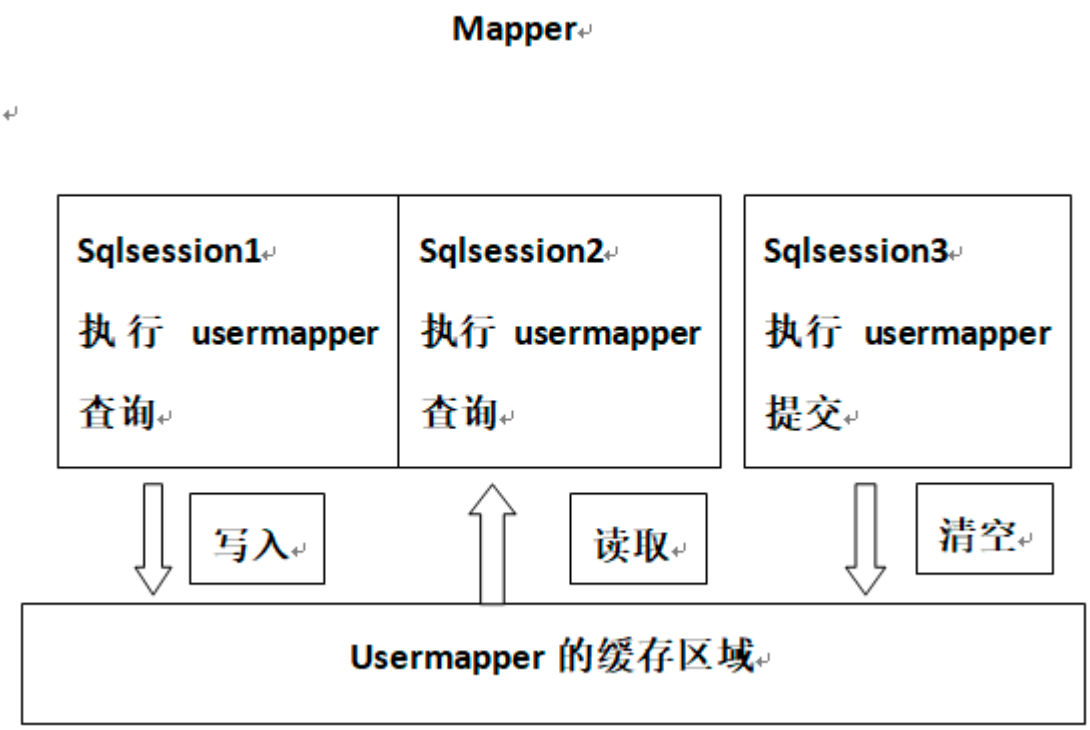
既然有了一级缓存，那么为什么要提供二级缓存呢?

二级缓存是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。二级缓存的作用范围更大。

还有一个原因，实际开发中，MyBatis通常和Spring进行整合开发。Spring将事务放到Service中管理，对于每一个service中的sqlsession是不同的，这是通过mybatis-spring中的org.mybatis.spring.mapper.MapperScannerConfigurer创建sqlsession自动注入到service中的。 每次查询之后都要进行关闭sqlSession，关闭之后数据被清空。所以spring整合之后，如果没有事务，一级缓存是没有意义的。

二级缓存：是mapper级别的缓存，多个SqlSession去操作同一个Mapper的sql语句，多个SqlSession可以共用二级缓存，二级缓存是跨SqlSession的。 UserMapper有一个二级缓存区域（按namespace分），其它mapper也有自己的二级缓存区域（按namespace分）。每一个namespace的mapper都有一个二级缓存区域，两个mapper的namespace如果相同，这两个mapper执行sql查询到数据将存在相同的二级缓存区域中。

二级缓存原理：



`

```java
@Test
public void testCache2() throws Exception {

 SqlSession sqlSession1 = sqlSessionFactory.openSession();

 SqlSession sqlSession2 = sqlSessionFactory.openSession();

 UserMapper userMapper1 = sqlSession1.getMapper(UserMapper.class);

 User user1 = userMapper1.findUserById(1);

 System.out.println(user1);

 sqlSession1.close();

 UserMapper userMapper2 = sqlSession2.getMapper(UserMapper.class);

 User user2 = userMapper2.findUserById(1);

 System.out.println(user2);

 sqlSession2.close();

}
```

执行结果：

```
DEBUG [main] - Cache Hit Ratio [com.iot.mybatis.mapper.UserMapper]: 0.0

DEBUG [main] - Opening JDBC Connection

DEBUG [main] - Created connection 103887628.

DEBUG [main] - Setting autocommit to false on JDBC Connection [com.mysql.jdbc.JDBC4Connection@631330c]

DEBUG [main] - ==> Preparing: SELECT * FROM user WHERE id=?

DEBUG [main] - ==> Parameters: 1(Integer)

DEBUG [main] - <==  Total: 1

User [id=1, username=张三, sex=1, birthday=null, address=null]

DEBUG [main] - Resetting autocommit to true on JDBC Connection [com.mysql.jdbc.JDBC4Connection@631330c]

DEBUG [main] - Closing JDBC Connection [com.mysql.jdbc.JDBC4Connection@631330c]

DEBUG [main] - Returned connection 103887628 to pool.

DEBUG [main] - Cache Hit Ratio [com.iot.mybatis.mapper.UserMapper]: 0.5

User [id=1, username=张三, sex=1, birthday=null, address=null]
```

我们可以从打印的信息看出，两个sqlSession，去查询同一条数据，只发起一次select查询语句，第二次直接从Cache中读取。

前面我们说到，Spring和MyBatis整合时， 每次查询之后都要进行关闭sqlSession，关闭之后数据被清空。所以spring整合之后，如果没有事务，一级缓存是没有意义的。那么如果开启二级缓存，关闭sqlsession后，会把该sqlsession一级缓存中的数据添加到namespace的二级缓存中。这样，缓存在sqlsession关闭之后依然存在。

总结：

对于查询多commit少且用户对查询结果实时性要求不高，此时采用mybatis二级缓存技术降低数据库访问量，提高访问速度。

但不能滥用二级缓存，二级缓存也有很多弊端，从MyBatis默认二级缓存是关闭的就可以看出来。

二级缓存是建立在同一个namespace下的，如果对表的操作查询可能有多个namespace，那么得到的数据就是错误的。

举个简单的例子：

订单和订单详情，orderMapper、orderDetailMapper。在查询订单详情时我们需要把订单信息也查询出来，那么这个订单详情的信息被二级缓存在orderDetailMapper的namespace中，这个时候有人要修改订单的基本信息，那就是在orderMapper的namespace下修改，他是不会影响到orderDetailMapper的缓存的，那么你再次查找订单详情时，拿到的是缓存的数据，这个数据其实已经是过时的。

根据以上，想要使用二级缓存时需要想好两个问题：

1）对该表的操作与查询都在同一个namespace下，其他的namespace如果有操作，就会发生数据的脏读。

2）对关联表的查询，关联的所有表的操作都必须在同一个namespace。