私塾在线《高级软件架构师实战培训阶段二》

——跟着cc学架构系列精品教程

模块间的相互调用-1

- 产生的问题描述
- Java中常见的远程调用方式：
  Socket、Http、TCP、UDP、RPC、RMI、JMS、WebService……
- 常见的框架介绍

1：Hessian：类似于RMI，使用二进制消息来进行远程调用。与RMI不同的是，它的二进制消息可以在非Java中使用，它实现了一种跨编程语言的对象序列化方法

2：Burlap：是一种基于XML的远程调用技术，但和其他基于XML的远程技术（如SOAP或XML-RPC）不同，Burlap的消息结构是尽可能的简单，不需要额外的外部定义语言（如WSDL）

3：Dubbo：阿里开源的分布式服务框架，通过高性能的RPC实现远程服务的调用，可以和Spring框架无缝集成，其架构类似于ESB。

4：Spring的HttpInvoker：类似于RMI，基于HTTP协议来进行远程调用，使用java的序列化机制，要求客户端和服务端都是基于Java的

5：WebService

模块间的相互调用-2

- 方案的选择

一：如果系统全部为内部可控的

1：量级不太大，可以考虑使用Hessian/ Burlap

2：量级较大，且交互要求较高，那么dubbo是一个现成、成熟的选择

缺点：需要很多额外的成本，比如学习成本，按需改进的成本等

3：交互要求并不高，主要是相互调用的需求，可以考虑自己实现

优点：完全按需定制，完全可控，升级、改进和完善都方便

缺点：需要投入开发成本，且完善成熟有一个过程

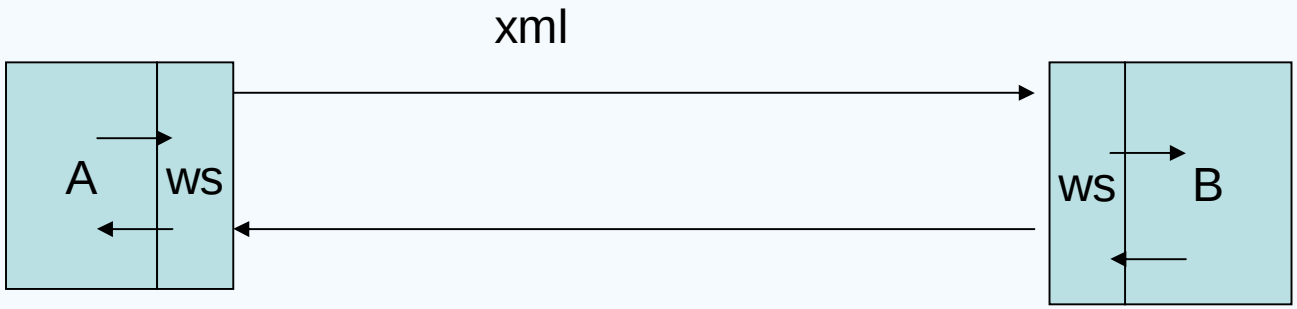
二：系统包含很多外部的应用，不能全部可控，且很多异构的系统

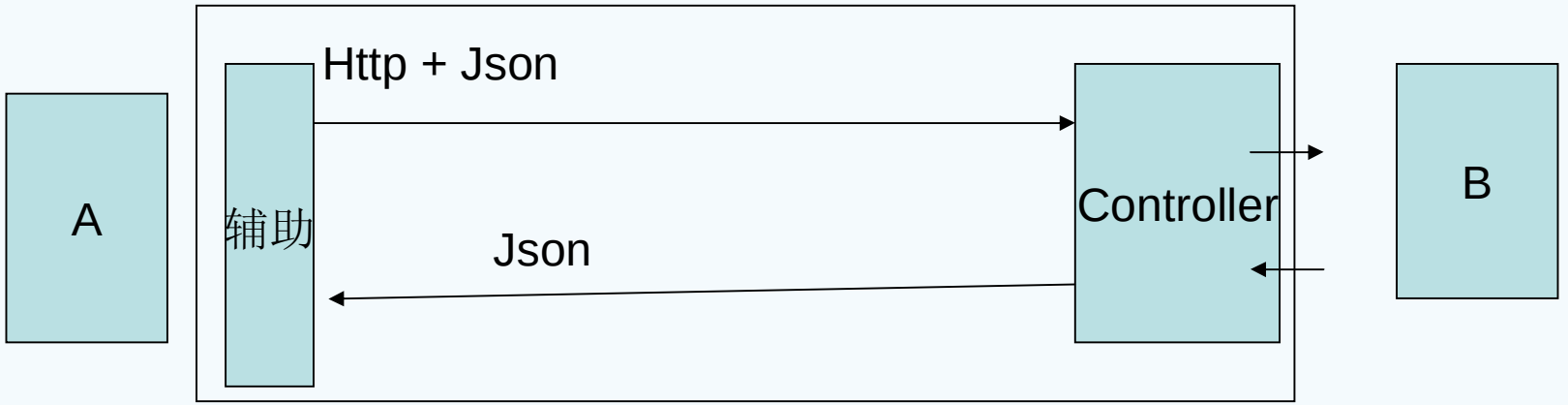
1：如果要求不是很复杂的话，WebService 是不错的选择

2：如果要求非常复杂，且涉及很多业务流，那就选择一个ESB平台

- 抛砖引玉：一种自己实现的简单方案（简化、高效，够用就好）

思路、核心部分的简单实现


xml
A
ws
ws
B


A
辅助
Http + Json
Json
Controller
B

模块间的相互调用-3

- 需要添加或调整的资源

1：添加处理Json的依赖包

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>1.1.36</version>
</dependency>
```

2：调整Goods模块的数据层处理部分，现在需要使用GoodsMapperDAO

3：在GoodsMgrWeb的Resource里面，添加GoodsMapper.xml

4：调整GoodsMgrWeb的applicationContext.xml，使其mapperLocations能够访问到GoodsMapper.xml

模块间的相互调用-4

- 更多需要考虑的问题

1：长连接，连接池，可以考虑HttpClient
2：高并发，多线程池，可以考虑使用apache的common-pool
3：快速的网络传输，可以考虑使用NIO，比如：Mina框架，Netty框架等
4：大数据量，数据压缩传输，可以考虑Java的GZip
5：可用性、稳定性、容错
6：分布式的事务
7：访问安全、数据安全等
8：服务的集群，服务的注册和管理等
……

统一会话管理-1

- 产生的问题描述
- 解决方案

1：根据IP或者Cookie来映射访问同一服务器，如：Nginx的IP_Hash，nginx-upstream-jvm-route等

2：采用统一的会话管理，可以把会话数据存放在公共的地方，比如Memcached

（1）自行实现

（2）结合框架去实现，比如使用Shiro

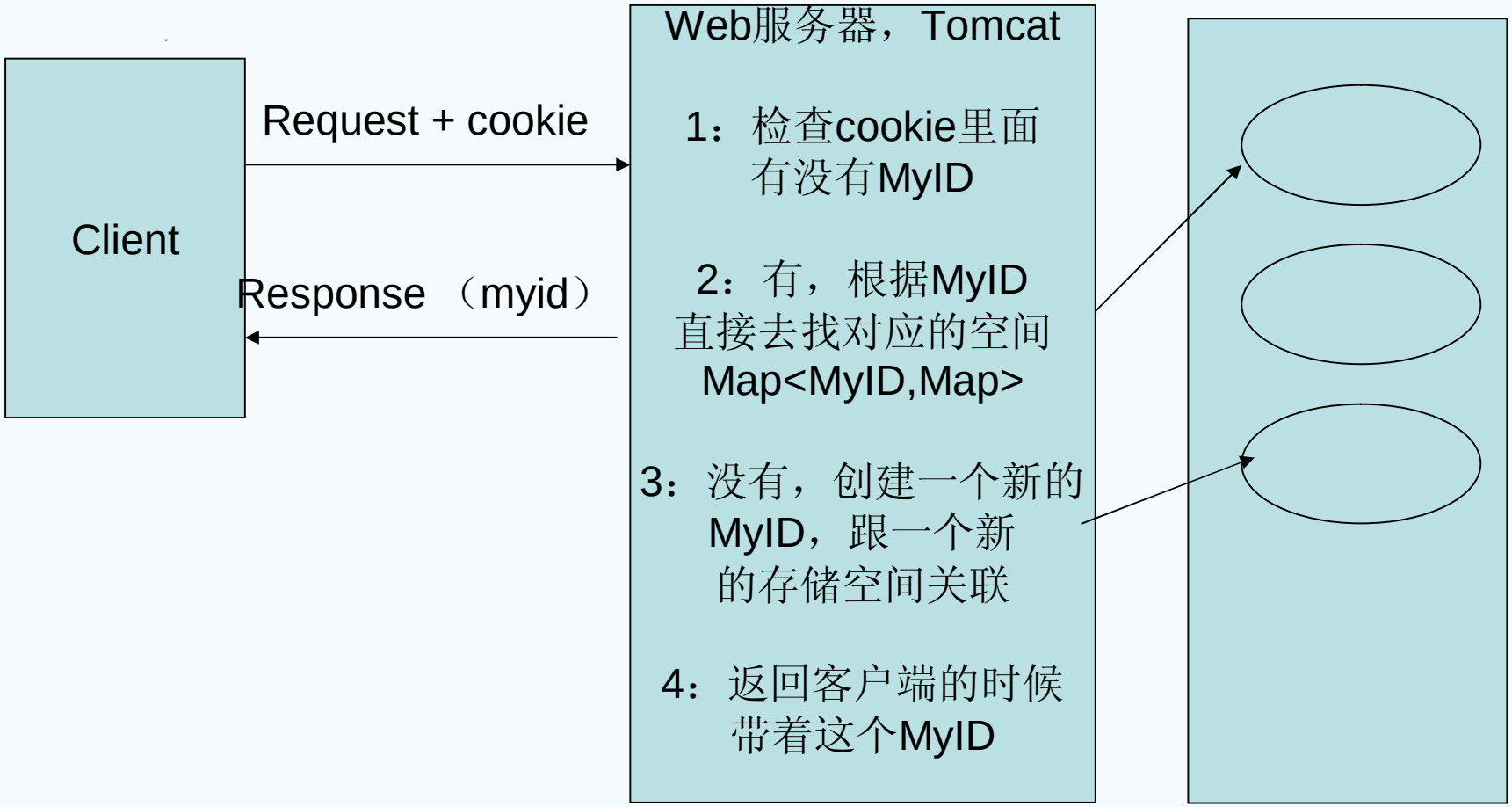
3：把会话序列化后，存放到客户端Cookie里面

- 方案的选择
- 需要添加或调整的资源

1：添加shiro需要的资源包

```
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-core</artifactId>
    <version>1.2.2</version>
</dependency>
```


Client
Request + cookie
Response （jsid）
Web服务器，Tomcat
1：检查cookie里面有没有JSessionID
2：有，根据JSessionID直接去找对应的空间
3：没有，创建一个新的JSessionID，跟一个新的存储空间关联
4：返回客户端的时候带着这个JSessionID


Client
Request + cookie
Response （myid）
Web服务器，Tomcat
1：检查cookie里面有没有MyID
2：有，根据MyID直接去找对应的空间Map<MyID,Map>
3：没有，创建一个新的MyID，跟一个新的存储空间关联
4：返回客户端的时候带着这个MyID

统一会话管理-2

```
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-web</artifactId>
    <version>1.2.2</version>
</dependency>
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-aspectj</artifactId>
    <version>1.2.2</version>
</dependency>
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-ehcache</artifactId>
    <version>1.2.2</version>
</dependency>
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-spring</artifactId>
    <version>1.2.2</version>
</dependency>
```

统一会话管理-3

2：在spring-mvc.xml中添加对shiro的filter的配置

```
<bean id="shiroFilter"
    class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
        <property name="securityManager" ref="securityManager"/>
        <property name="loginUrl" value="/toLogin"/>
        <property name="unauthorizedUrl" value="/unauthorized.jsp"/>
        <property name="filterChainDefinitions">
            <value>
                /jcaptcha* = anon
                /logout = anon
            </value>
        </property>
    </bean>
```

3：添加Shiro本身的配置文件applicationContext-shiro.xml

4：开发自定义的Shiro的SessionDAO

统一会话管理-4

5：在web.xml中添加对shiroFilter的配置

```
<filter>
        <filter-name>shiroFilter</filter-name>
        <filter-
class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
        <init-param>
                <param-name>targetFilterLifecycle</param-name>
                <param-value>true</param-value>
        </init-param>
</filter>
<filter-mapping>
        <filter-name>shiroFilter</filter-name>
        <url-pattern>/*</url-pattern>
</filter-mapping>
```

6：在web.xml中，修改contextConfigLocation的值为：

```
classpath:applicationContext*.xml,classpath*:spring-mvc.xml
```

统一会话管理-5

7：在LoginController的登录中，添加Session信息：
（1）注入SecurityManager

```
@Autowired
private org.apache.shiro.mgt.SecurityManager sm = null;
```

（2）向Session中添加数据，比如：

```
SecurityUtils.setSecurityManager(sm);
Subject currentUser = SecurityUtils.getSubject();
currentUser.getSession().setAttribute("LOGIN_USER", cm);
```

8：在IndexController或者其他需要使用Session的地方，同样是先注入，然后获取到Shiro提供的Session对象，再从里面获取数据

9：对Front工程，在pom.xml中注掉filemgr的依赖，在web.xml中注掉QueueReceiver的配置，从Mapper文件夹下面，去掉FileMapper.xml

10：别忘记了开启服务器上的Memcached，会话数据要存放到memcached里面：

```
./memcached -d -m 10 -u root -l 192.168.1.106 -p 2222 -c 256 -P /tmp/memcached.pid
```

统一会话管理-6

- 更多的问题

1：如果用户关闭了Cookie
2：Cookie数据的安全性
3：跨域访问Cookie
4：公共缓存的规划、集群和数据维护

单点登录

- 产生的问题描述

- “伪”在何处

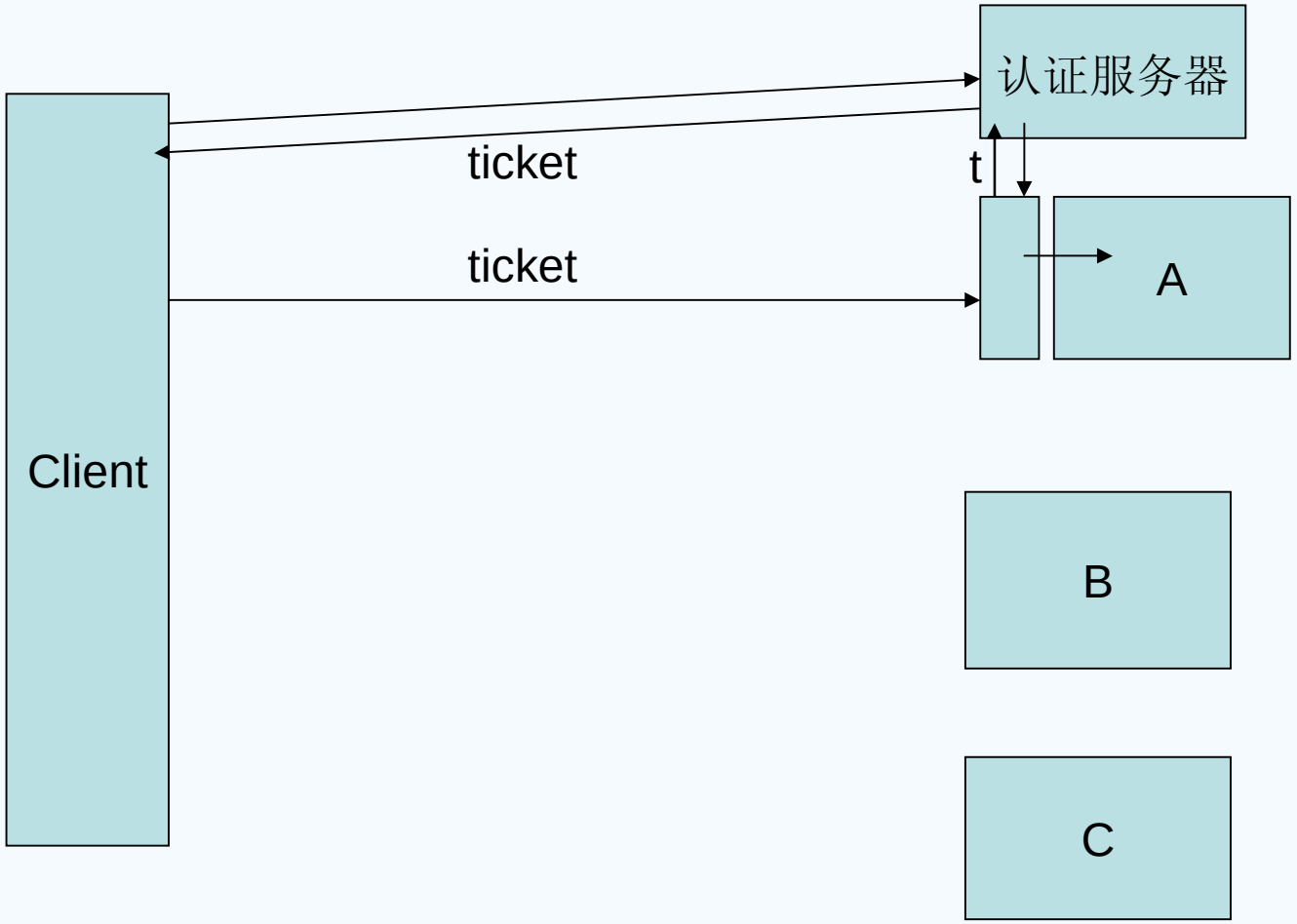
跟EAI中的SSO相比，这里所说的单点登录是很简单的，算不上是“真正”的SSO

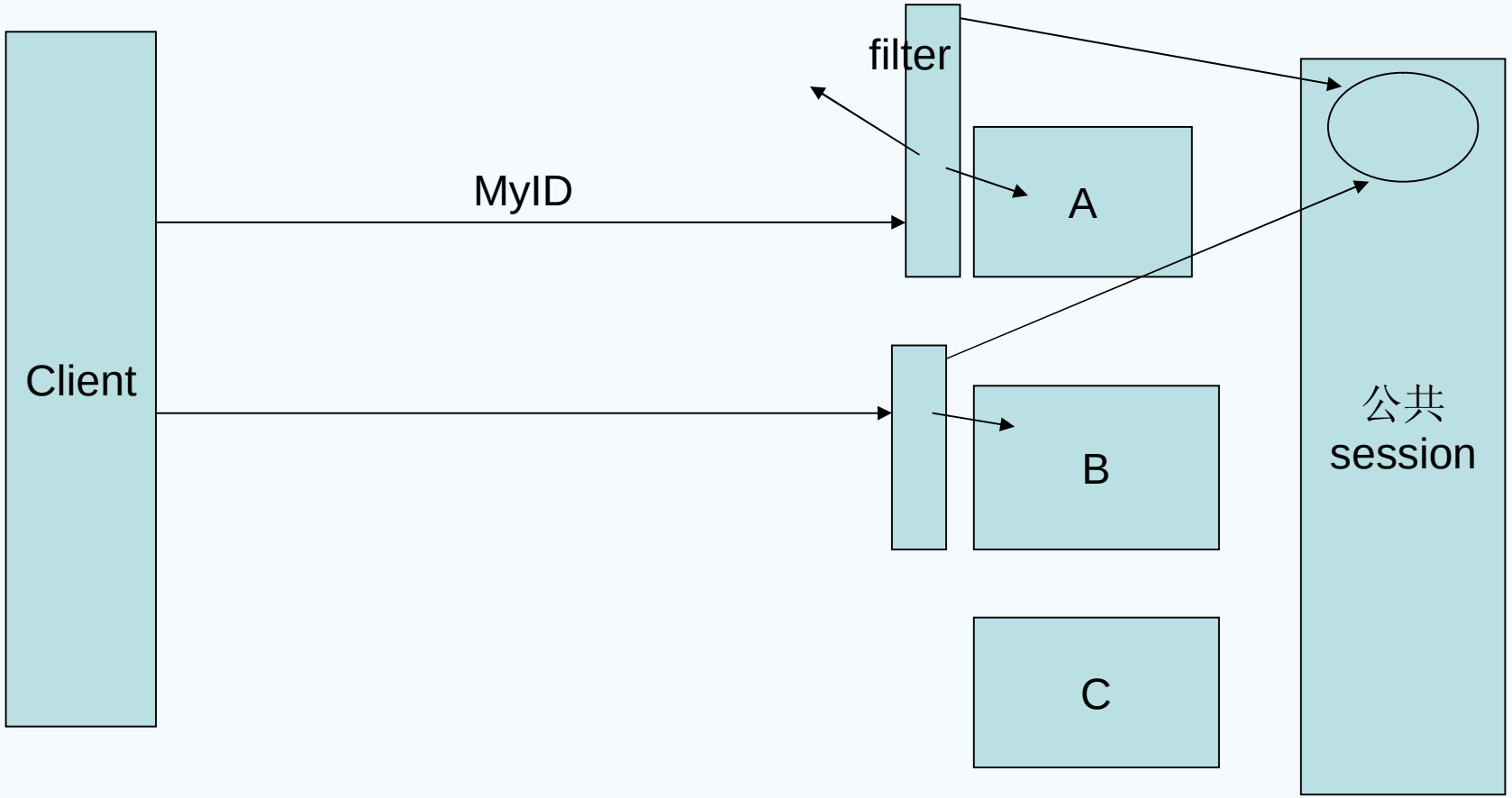
1：本身就是一个系统，只有一套用户和权限系统
2：对用户的验证方式是统一的
3：都是内部系统，相互信任，所以也就不用验证是否可访问系统了

- 解决方案

1：简单的：使用Shiro的统一会话管理，实现单点登录
2：稍麻烦些的：使用Shiro+CAS来实现
3：更麻烦的：使用专业的SSO框架或产品

- 方案选择


认证服务器
ticket
t
ticket
A
Client
B
C


filter
MyID
A
Client
公共
session
B
C

一致性更新-1

- 产生的问题描述

- 分布式的一致性介绍

  对于一致性，可以分为从客户端和服务端两个不同的视角。从客户端来看，一致性指的是并发访问时更新过的数据如何获取的问题；从服务端来看，则是更新的数据如何复制分布到整个系统，以保证数据最终一致。

  一致性是有并发读写才有的问题，因此在理解一致性的问题时，一定要注意结合考虑并发读写的场景。

- CAP的最终一致性

  从客户端角度，并发访问时，更新过的数据在不同进程如何获取的不同策略，决定了不同的一致性。

  对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是强一致性；如果能容忍后续的部分或者全部访问不到，则是弱一致性；如果经过一段时间后要求能访问到更新后的数据，则是最终一致性。

一致性更新-2

- 常见的解决方案

一：有一个公共的数据库

1：单点部署，也就是整个系统中只有一个地方能修改这个数据

2：采用版本控制

二：分散到多个数据库

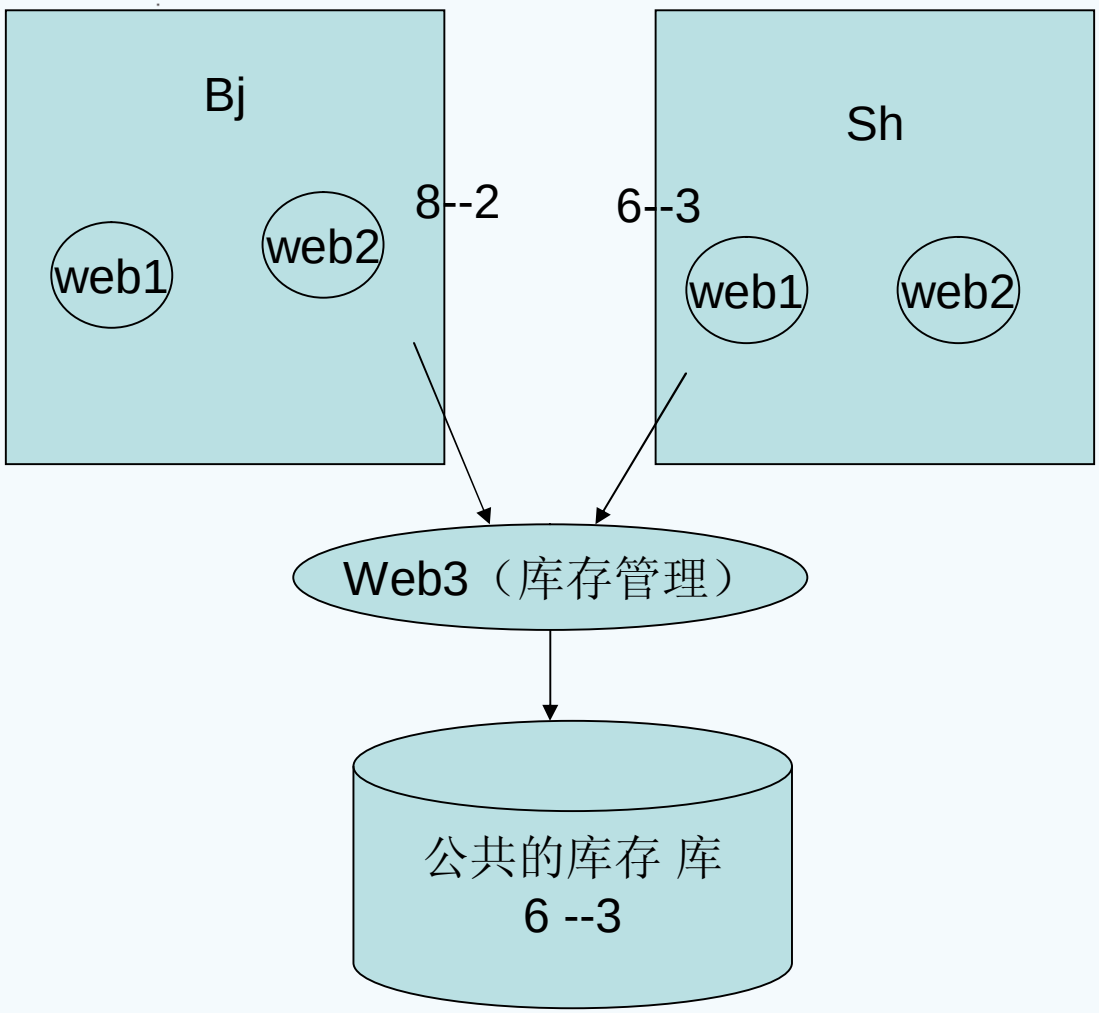
1：可以把问题简化成为只有一个数据库的情况

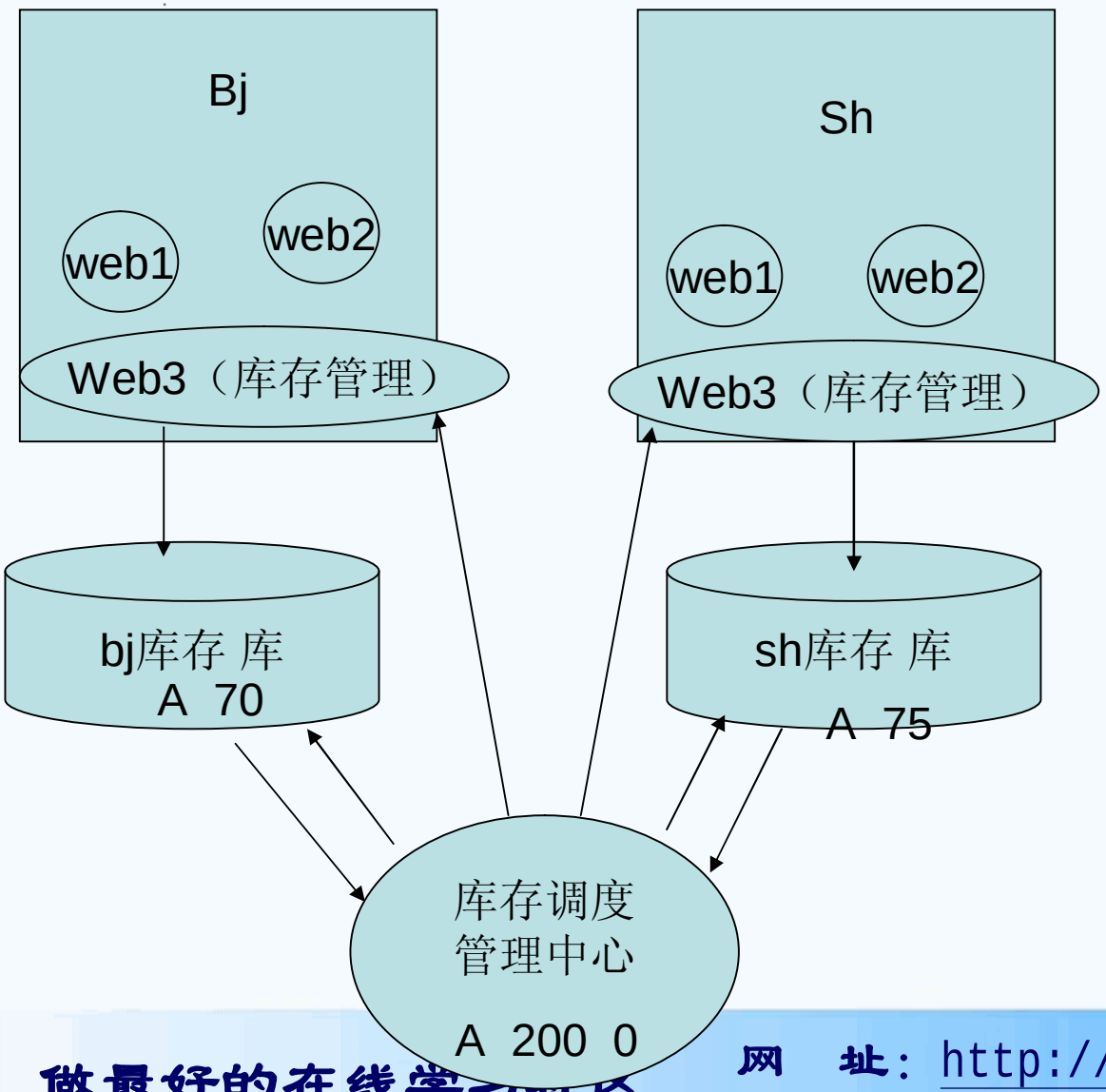
2：采用预分配数据，动态进行逻辑调整

- 方案的选择

- 重要的提示

就是一定要结合着具体的部署方案，以及具体的业务功能，还有具体使用的场景，进行综合分析和思考，去设计最合理的解决方案。


Bj
web1
web2
8--2
Sh
6--3
web1
web2
Web3（库存管理）
公共的库存 库
6 --3


Bj
web1
web2
Web3（库存管理）
Sh
web1
web2
Web3（库存管理）
bj库存 库
A 70
sh库存 库
A 75
库存调度
管理中心
A 200 0

50

分布式事务-1

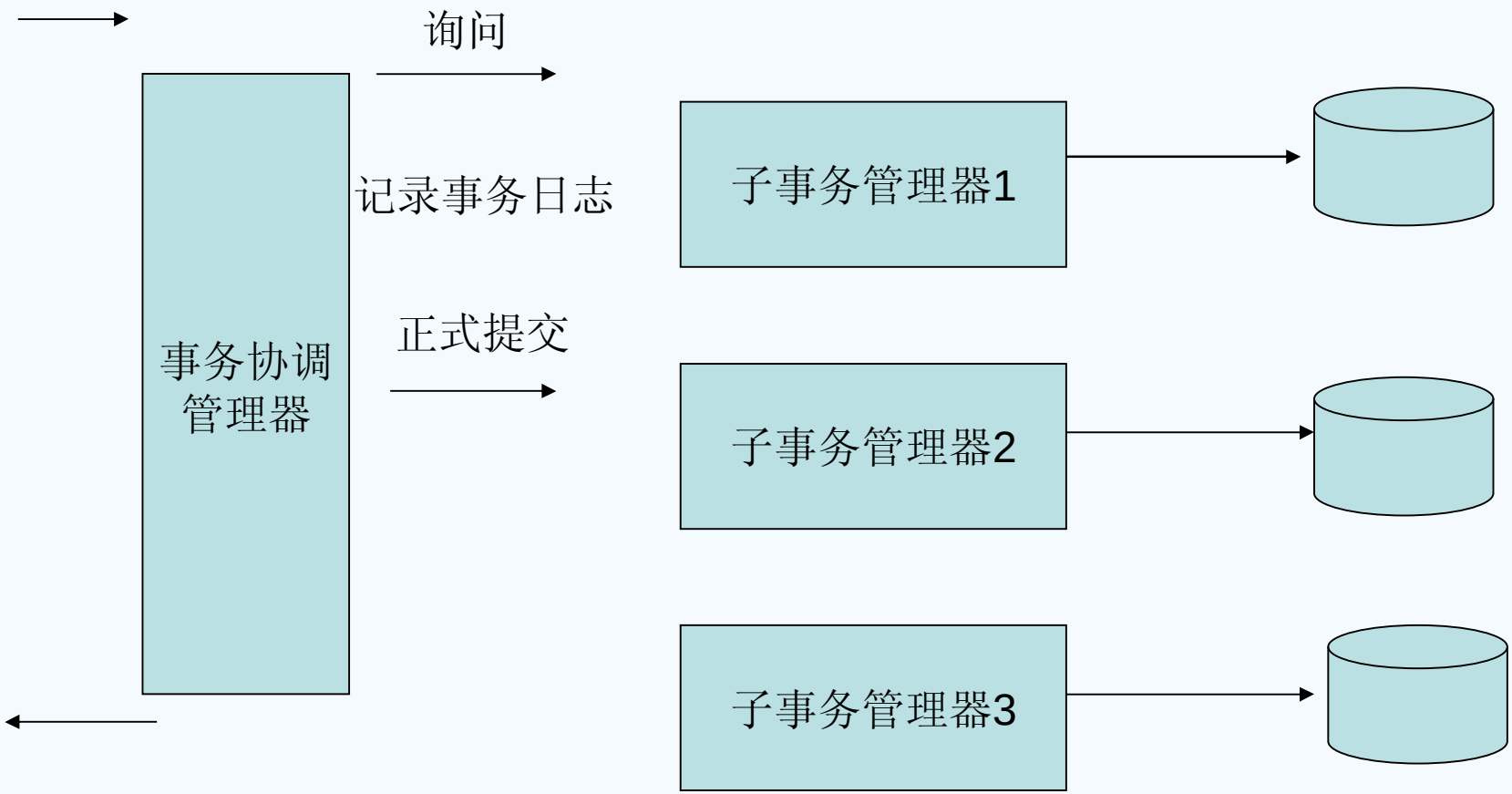
- 产生的问题描述

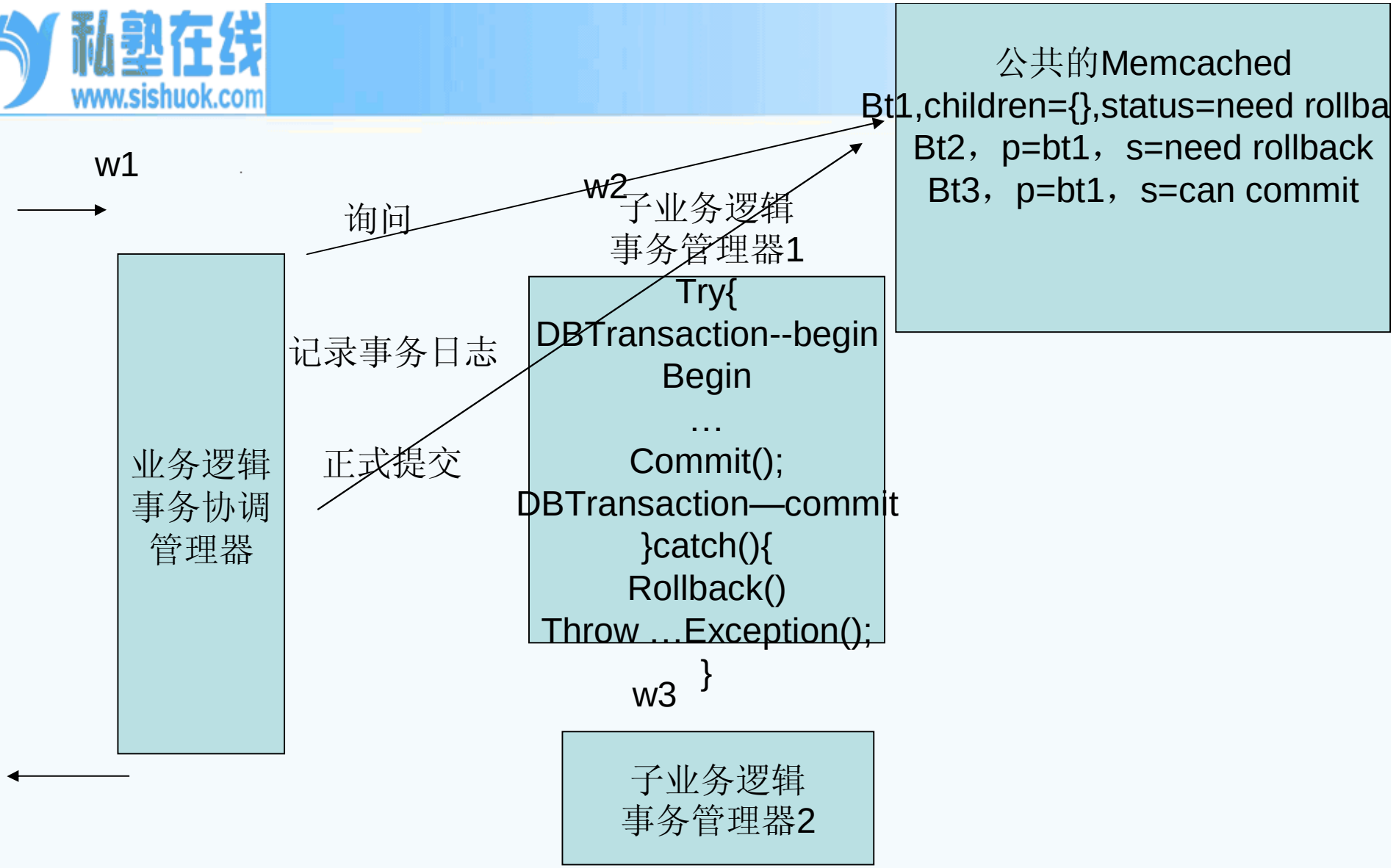
- 解决方案

1：同一个Web服务器，多个数据库，可以使用Atomikos
2：跨越多个Web服务器的事务，如果远程调用支持事务传播，那么使用JTA就可以；如果不支持事务传播，就尽量转化为一个web服务器的情况
3：自行开发事务逻辑事务管理器
4：采用业务补偿回滚的方式
5：重新设计和规划

- 方案的选择

- Try{
- 1：库存+20=== w1
- 2：写入订单 === w2
- 3：改写价格 === w3
- }catch(){
- //判断出是第3步失败了
  //回滚1 继续调w1，库存-20
  //回滚2 调w2，删除订单
- }


询问
事务协调管理器
记录事务日志
正式提交
子事务管理器1
子事务管理器2
子事务管理器3


公共的Memcached
Bt1,children={},status=need rollba
Bt2，p=bt1，s=need rollback
Bt3，p=bt1，s=can commit
w1
询问
w2
子业务逻辑
事务管理器1
Try{
DBTransaction--begin
Begin
…
Commit();
DBTransaction—commit
}catch(){
Rollback()
Throw …Exception();
}
记录事务日志
正式提交
业务逻辑
事务协调
管理器
w3
子业务逻辑
事务管理器2

- A类里面
- Try{
- mainBTUuid = Bt.begin(“puuid”);
- 调用w1的b业务, mainBTUuid 传过去
- 调用w2的c业务mainBTUuid 传过去

自己操作数据库

```
Bt.commit();
}catch(){
    Bt.rollback();
}
```

分布式事务-2

- 使用Atomikos

1：添加需要使用的lib

```
<dependency>
        <groupId>com.atomikos</groupId>
        <artifactId>transactions-jdbc</artifactId>
        <version>3.9.3</version>
</dependency>
<dependency>
        <groupId>com.atomikos</groupId>
        <artifactId>transactions-hibernate3</artifactId>
        <version>3.9.3</version>
</dependency>
<dependency>
        <groupId>javax.transaction</groupId>
        <artifactId>jta</artifactId>
        <version>1.1</version>
</dependency>
```

分布式事务-3

2：在Spring的配置文件中配置数据源

```
<bean id="jtaTransactionManager"
   class="org.springframework.transaction.jta.JtaTransactionManager">
   <property name="transactionManager">
        <bean class="com.atomikos.icatch.jta.UserTransactionManager"
   init-method="init" destroy-method="close">
              <property name="forceShutdown" value="true"/>
        </bean>
   </property>
   <property name="userTransaction">
        <bean class="com.atomikos.icatch.jta.UserTransactionImp">
      <property name="transactionTimeout" value="300" />
    </bean>
   </property>
</bean>
```

分布式事务-4

```
<bean id="jtaDataSource1" class="com.atomikos.jdbc.AtomikosDataSourceBean" init-method="init"
    destroy-method="close">
    <property name="uniqueResourceName" value="ds1"/>
    <property name="xaDataSourceClassName"
    value="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"/>
    <property name="xaProperties">
          <props><prop
    key="url">jdbc:mysql://localhost:3306/arch1?useUnicode=true&amp;characterEncoding=UTF-
    8</prop>
                    <prop key="user">root</prop>
                    <prop key="password">cc</prop>
                    <prop key="pinGlobalTxToPhysicalConnection">true</prop>
          </props>
    </property>
    <property name="minPoolSize" value="10" />
    <property name="maxPoolSize" value="100" />
    <property name="borrowConnectionTimeout" value="30" />
    <property name="testQuery" value="select 1" />
    <property name="maintenanceInterval" value="60" />
</bean>
```

3：添加事务的配置applicationContext-tx.xml

4：写测试方法进行测试

分布式事务-5

- 重要的提示

  分布式事务不但复杂，对系统性能也有极大影响，因此能够规避就尽量规避，在一开始设计的时候，就不要出现需要分布式事务的场景。如果不可避免，也尽量采用分布代价最小的方案

高可用性(HA)

- 产生的问题描述

- 解决方案
  可以使用Keepalived/Heartbeat等类似的软件

HA简介-1

- 什么是HA

HA(High Available)，高可用性群集，指的是通过一组计算机系统提供透明的冗余处理能力，从而保证系统服务高度的连续可用。

- 几点说明

1：HA通常是软件和硬件相结合的集群方案，是自动且透明的
2：只有硬件的方案不是HA，那是热备，通常是人工的切换备用机
3：HA通常由软件检测故障，一旦故障发生立即切换服务到集群中正常的服务上，通过提供故障恢复，实现最大化系统和应用的可用性
4：HA在故障恢复的切换过程中，会有短暂的服务暂停的过程，因为选举新的服务器，以及资源转移都需要一定的时间，当然这个时间很短
5：HA的衡量指标通常有：平均无故障时间(MTTF)，平均维修时间(MTTR)，可用性=MTTF/(MTTF+MTTR)

HA简介-2

- HA的几种常见部署模式

1：主从方式：两台服务器，一台为主，另外一台为备份服务器
2：对称方式：两台服务器，互为备份
3：多机方式：多台服务器，故障时切换至其中一台

- HA的基本实现原理

1：提供虚拟IP给外部访问
2：节点之间通过心跳或信息报文来确定健康状态
3：节点之间通讯通常会加密，以防止非法主机加入
4：当前提供服务的机器出现问题后，需要按照一定的规则，投票选举出新的提供服务的机器，并接管服务