

EJB 성능 향상을 위한 Best Practices 1부

Document Information

Document title:	EJB 성능 향상을 위한 Best Practices 1부
Document file name:	EJB_Performance1_jstorm_1.0_final
Revision number:	<1.0>
Issued by:	<전부현, 이용원>
Issue Date:	<2002/2/24>
Status:	final

Content Information

Audience	EJB 중급
Abstract	이 섹션에서는 다음과 같은 타입의 EJB를 사용할 때 퍼포먼스를 향상시키는 방법을 중점적으로 다룰 것이다. ① 무상태 세션 빈 ② 상태유지 세션 빈 ③ 엔티티 빈 ④ 메시지 드리븐 빈 이 섹션에서는 EJB 1.1 과 EJB 2.0 스펙을 다룰 것이다. 먼저 빈의 일부 또는 모든 타입들 사이의 공통적인 튜닝 과정을 설명하고, 그리고 나서 각각의 빈 타입들에 대해 설명할 것이다.
Reference	http://www.precisejava.com/javaperf/j2ee/EJB.htm
Benchmark information	

Table of Contents

EJB 성능 향상을 위한 Best Practices.....	4
1. EJB의 개관	4
2. EJB의 공통적인 튜닝 과정	6
2.1 EJB와 비-EJB 사이의 선택.....	6
2.2 로컬 EJB와 원격 EJB 중 하나를 선택	7
2.3 pass-by-reference를 이용한 최적화	9
2.3 엔티티 빈들을 세션 빈으로 래핑	9
2.5 큰 단위(coarse granularity)를 이용해 메소드 호출을 감소시켜라	11
2.6 원격 EJB의 직렬화를 컨트롤하라	12
2.7 EJBHome 객체 레퍼런스를 캐싱하라	13
2.8 트랜잭션을 컨트롤하라	15
2.9 최적의 트랜잭션 타임아웃을 설정하라	16
2.10 확장성을 위하여 클러스터링을 이용하라	17
2.11 쓰레드 숫자를 튜닝하라	18
2.12 HttpSession과 상태유지 세션 빈 중 하나를 선택하라	18
2.13 EJB 서버를 선택하라	19
3. 무상태 세션 빈을 튜닝하라.....	21
3.1 무상태 세션 빈 라이프 사이클	21
3.2 무상태 세션 빈의 인스턴스 풀 사이즈를 튜닝하라	23
3.3 setSessionContext() 또는 ejbCreate() 메소드를 캐싱하여 사용하라	24
3.4 ejbRemove() 메소드에서 리소스를 해제하라.....	24
4. 상태유지 세션 빈을 튜닝하라.....	25
4.1 상태유지 세션 빈의 라이프 사이클	25
4.2 상태유지 세션 빈의 인스턴스 캐시 사이즈를 튜닝하라	28
4.3 상태유지 세션 빈을 위해 최적화된 빈 연령을 설정하라	28
4.4 상태유지 세션 빈의 직렬화를 컨트롤하라	28
4.5 상태유지 세션 빈을 명시적으로 제거하라	29



EJB 성능 향상을 위한 Best Practices

1. EJB의 개관

엔터프라이즈 자바빈즈는 다음과 같은 서비스를 제공함으로써 개발자의 작업을 간소화 해주는 컴포넌트를 말한다.

- ① RMI/RMI-IIOP (분산 컴퓨팅)
- ② 트랜잭션
- ③ 보안
- ④ 리소스 풀링
- ⑤ 쓰레드 관리
- ⑥ 클라이언트 상태 관리
- ⑦ 영속성
- ⑧ 메시징
- ⑨ 라이프 사이클 등등.

개발자는 이러한 서비스를 위한 기반 코드를 작성할 필요가 없으며, 벤더가 이러한 서비스를 지원하는 EJB 컨테이너/서버를 제공한다. 개발자는 코드의 작성 없이 간단하게 이러한 서비스를 이용할 수 있다. 그러나 이러한 모든 서비스들은 공짜가 아니며, 따라서 가격을 지불해야 한다. 그러므로 애플리케이션을 작성하기 전에 먼저 자신이 가진 자원들과 그에 따른 트레이드오프의 모든 측면을 신중하게 따져봐야 한다.

이 섹션에서는 다음과 같은 타입의 EJB를 사용할 때 퍼포먼스를 향상시키는 방법을 중점적으로 다룰 것이다.

- ① 무상태 세션 빈
- ② 상태유지 세션 빈
- ③ 엔티티 빈
- ④ 메시지 드리븐 빈

이 섹션에서는 EJB 1.1 과 EJB 2.0 스펙을 다룰 것이다. 먼저 빈의 일부 또는 모든 타입들 사이의 공통적인 튜닝 과정을 설명하고, 그리고 나서 각각의 빈 타입들에 대해 설명할 것이다.

Note : 이 섹션은 독자가 EJB에 대한 기본 지식을 가지고 있다고 가정한다.

2. EJB의 공통적인 튜닝 과정

여기서는 엔터프라이즈 자바빈즈의 일부 또는 모든 타입들 사이의 공통적인 튜닝 과정을 설명한다.

2.1 EJB와 비-EJB 사이의 선택

먼저 자신에게 질문해야 하는 사항은 트랜잭션, RMI/RMI-IIOP, 보안, 클라이언트 상태 관리, 영속성 등과 같은 서비스를 사용하는 EJB는 비용과 연관됨에도 불구하고 왜 일반 자바 객체가 아닌 EJB를 사용하느냐 하는 것이다. 이러한 서비스를 애플리케이션에서 사용하기를 원한다면 EJB는 최선의 선택이겠지만, 그렇지 않다면 일반 자바 객체는 더 나은 선택이 될 것이다. 그리고 또한 이러한 선택은 애플리케이션에 얼마나 많은 서비스가 필요한가에 달려있다. 만약 트랜잭션 관리를 원한다면 EJB 보다는 JTA(Java Transaction API)를 사용할 수 있으며, 분산 컴퓨팅이 필요한 애플리케이션을 개발할 경우 EJB가 아닌 RMI/RMI-IIOP만으로도 충분하다.

EJB는 컴포넌트 아키텍처에 의해 지원되는 일반 자바 객체와는 다른 재사용 가능한 컴포넌트이며, 어떠한 EJB 호환 서버에도 재사용될 수 있는 EJB를 배포하는 것이 가능하다.

다음의 그림은 클라이언트가 EJB와 통신하는 방법에 대한 기반 기술을 나타내고 있다.

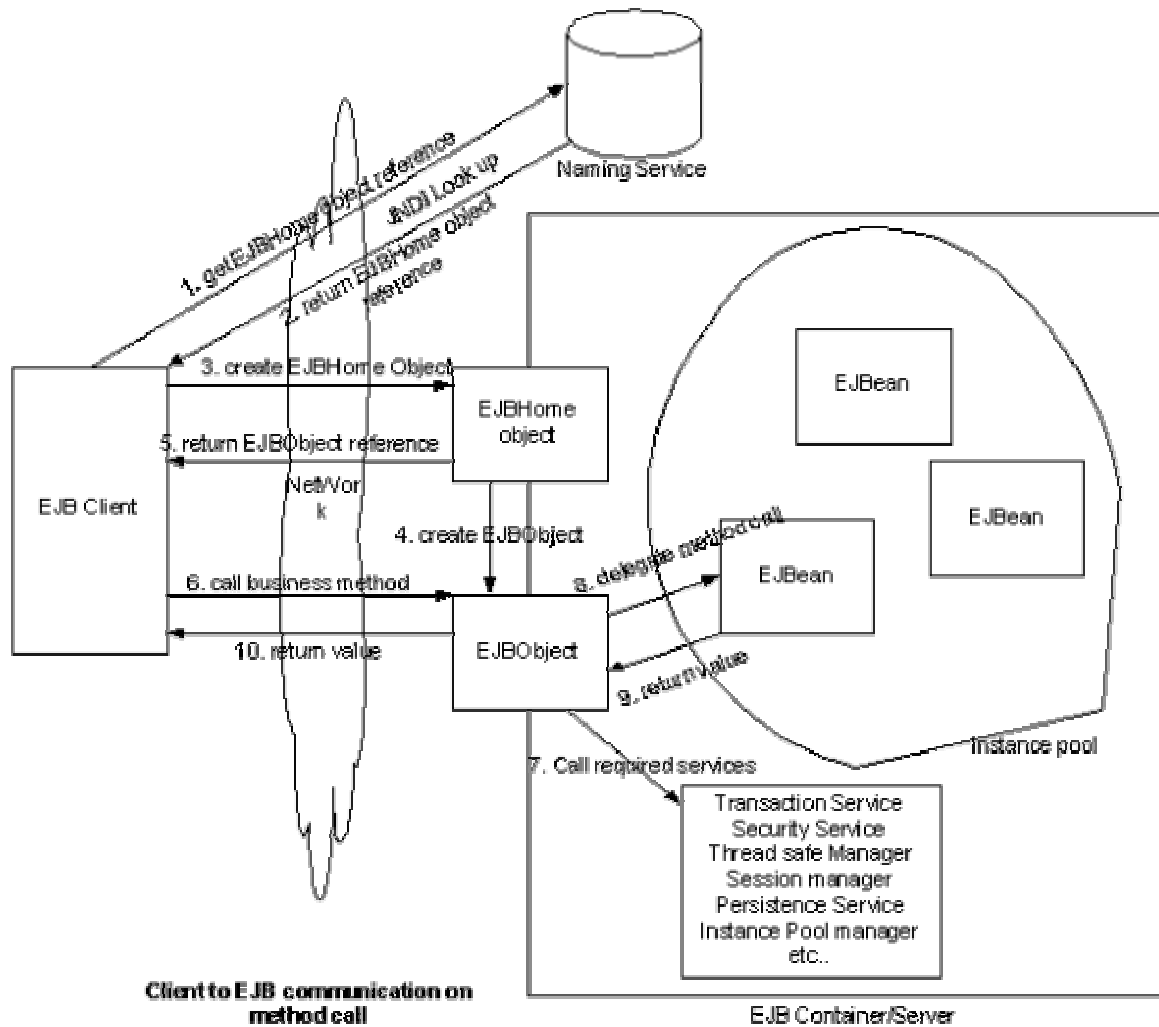


그림1 : 클라이언트와 EJB간의 통신

이 그림은 클라이언트와 EJB간의 통신이 이루어지는 방법을 설명하고 있으며, 이러한 전반적인 과정은 비용이 많이 든다.

그러므로 EJB의 일부 또는 모든 서비스가 필요하지 않을 경우에는 자바 객체를 사용하고, 그렇지 않을 경우에만 EJB를 선택하라.

2.2 로컬 EJB와 원격 EJB 중 하나를 선택

EJB 애플리케이션을 사용하기로 결정했으면, 먼저 로컬 EJB와 원격 EJB 중 하나를 선택해야 한다. EJB 1.1이 원격 EJB만을 지원하는데 반해 EJB 2.0은 분산 컴퓨팅 능력(RMI/RMI-IIOP)이 없이 로컬 EJB로 동작하는 로컬 인터페이스의 개념을 소개하고 있다.

원격 EJB는 리모트 인터페이스의 확장을 의미한다. 따라서 클라이언트는 EJB와 모든 메소드 호출에 대해 직렬화, 마샬링/언마샬링 RMI/RMI-IIOP를 이용해 통신하게 되고, 이는 퍼포먼스를 떨어뜨리게 된다.

EJB 클라이언트와 EJB를 같은 서버의 같은 JVM에 디플로이할 경우에는 원격 EJB가 필요하지 않으며, 로컬 EJB를 사용할 수 있다. 로컬 EJB는 통신을 위해 RMI/RMI-IIOP를 사용하지 않으며, 이러한 오버헤드를 일반 자바 메커니즘의 pass-by-reference를 이용하여 우회한다. 따라서 로컬 EJB는 원격 EJB에 비교하여 더 빠르며 효율적이다.

다음은 로컬 EJB와 원격 EJB의 차이점을 기술한 그림이다.

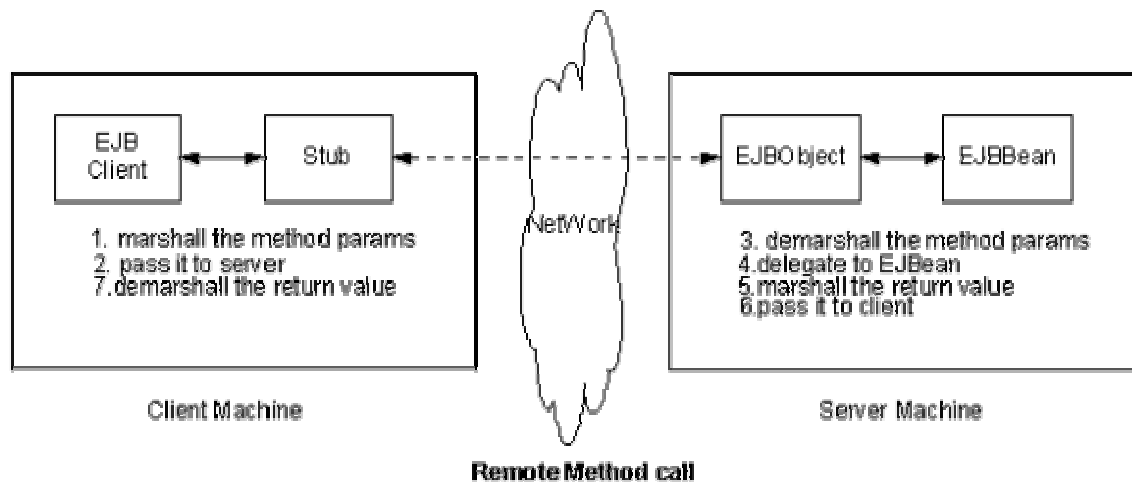


그림 2 : 원격 EJB

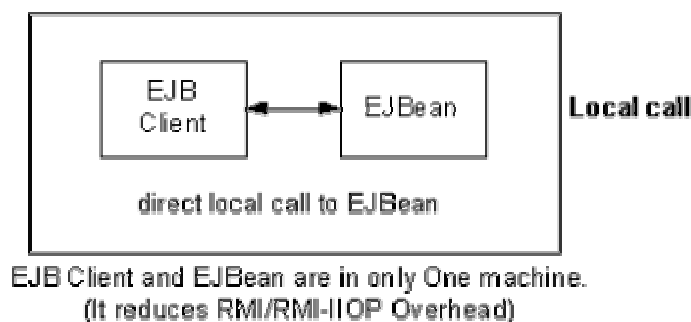


그림 3 : 로컬 EJB

EJB 2.0에서는 클라이언트와 EJB를 같은 머신/서버에 디플로이 하는 경우 RMI/RMI-IIOP 오버헤드를 줄이고 퍼포먼스를 향상시키기 위해 로컬 인터페이스를 사용하여 로컬 EJB로 사용하라.

2.3 pass-by-reference를 이용한 최적화

EJB 1.1은 EJB 2.0과는 달리 로컬 인터페이스를 지원하지 않으며, 따라서 모든 객체를 원격 객체로 사용할 수 밖에 없다. 클라이언트와 엔터프라이즈 빈을 같은 머신에 디플로이하기를 원하는 경우에도, 객체들이 로컬로 사용될 수 있음에도 원격으로 사용되는 오버헤드를 피할 수는 없다. 그러나 어떤 EJB 서버의 경우에는 디플로이먼트 디스크립터를 설정함으로써 이러한 문제를 해결할 수 있다. 예를 들어, 웹로직 서버의 `weblogic-ejb-jar.xml` 파일은 엘리먼트가 `by-reference`로 넘겨질 것인지 또는 `by-value`로 넘겨질 것인지를 정의하는 엘리먼트를 가지고 있다. 이러한 엘리먼트는 다음과 같다.

```
<enable-call-by-reference > true</enable-call-by-reference >
```

이 엘리먼트는 디폴트로 `true`로 설정되어 있다.

JBoss 서버에서 `jboss.xml` 파일은 다음과 같은 엘리먼트를 가지고 있다.

```
<container-invoker-conf>  
    <optimized>true</optimized>  
</container-invoker-conf>
```

어떤 벤더들은 디폴트로 `pass-by-reference`를 지원하지 않을 수 있으므로, 자세한 정보와 설정을 위해서는 현재 사용중인 서버의 문서를 참조하라.

EJB 1.1에서는 퍼포먼스의 향상을 위해 클라이언트와 EJB를 같은 머신/서버에 디플로이 하는 경우, EJB를 로컬 EJB로 사용하기 위해 벤더 의존적인 디플로이먼트 디스크립터 파일의 `pass-by-reference` 프로퍼티를 설정하라.

2.3 엔티티 빈들을 세션 빈으로 래핑

자바 스윙이나 JSP 또는 서블릿에서 엔티티 빈을 호출하면, 네트워크 오버헤드를 증가시키는 다수의 원격 네트워크 호출이 발생한다. 이러한 문제는 엔티티 빈들을 세션 빈으로 래핑함으로써 해결할 수 있다. 최선의 방법은 엔티티 빈들을 로컬 EJB로 만들고 세션 빈으로 래핑하는 것이다. 일반적으로 많은 수의 엔티티 빈들에 대한 호출이 필요한 요청을 수행하기 위하여, 예를 들어 클라이언트가 계좌 정보와 함께 주문을 표시하기를 원하는 경우에는 `PersonEntityBean`,

OrderEntityBean, 그리고 AccountEntityBean 등과 같은 세 엔티티 빈들을 호출해야 한다. 여기서 세 원격 호출이 발생하며, 이러한 세 엔티티 빈들을 OrderManager 세션 빈으로 래핑함으로써 원격 호출의 수를 최소화할 수 있으며, 이러한 경우 한번의 원격 호출만이 필요하게 된다. 세션 빈은 이러한 요청을 받아 세 번의 호출을 로컬에서 실행하고 값들을 리턴하는 과정을 한번의 호출에서 실행한다. 엔티티 빈을 로컬 빈으로 사용하기 위해서는 EJB 버전에 따라 앞에서 설명한 방법을 이용할 수 있다.

다음은 이 과정을 나타낸 그림이다.

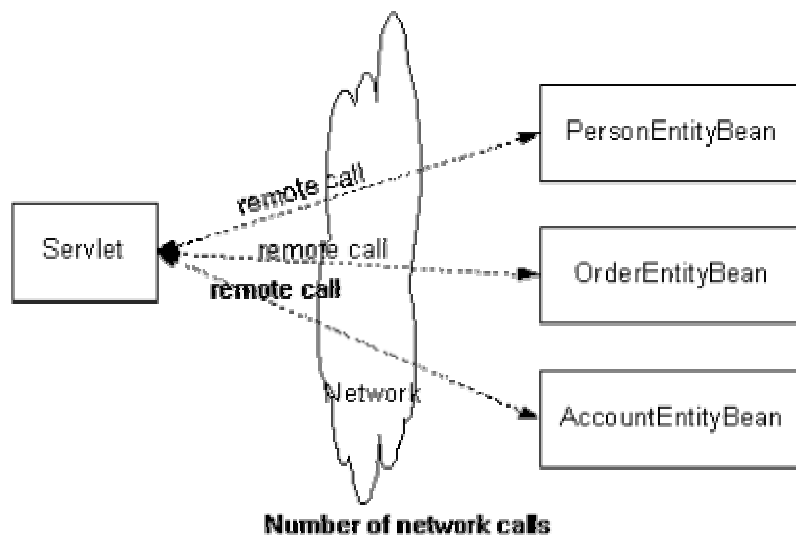


그림 4 : 네트워크를 통해서 직접 엔티티 빈을 호출

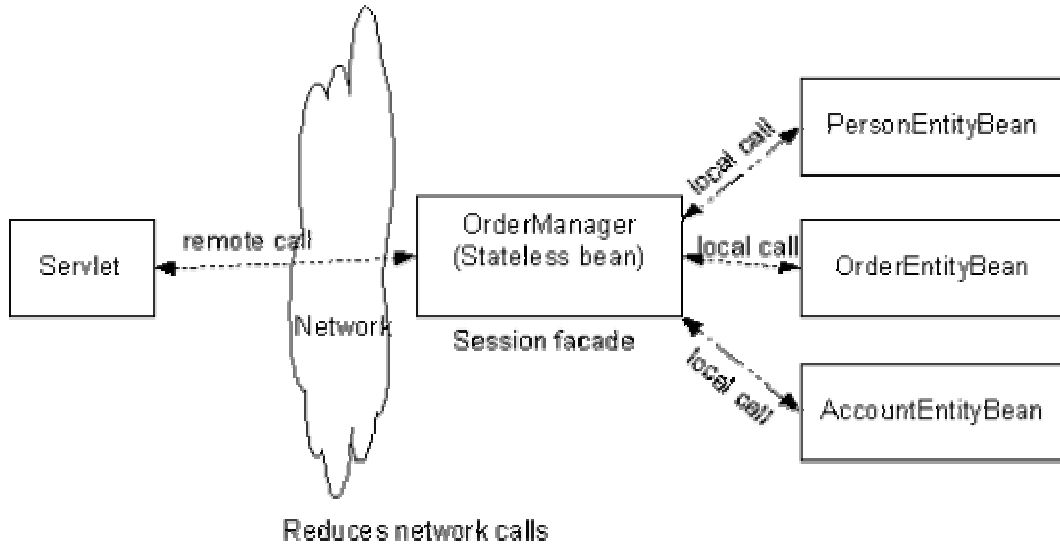


그림 5 : 네트워크 호출을 줄이기 위해서 엔티티 빈을 세션 빈으로 래핑

네트워크 호출을 줄이고 퍼포먼스를 향상시키기 위해서, 엔티티 빈들을 세션 빈으로 래핑하라. 이 기술은 퍼포먼스를 향상시킬 뿐만 아니라 좋은 선언적 트랜잭션 지원 및 디자인을 제공한다.

2.5 큰 단위(coarse granularity)를 이용해 메소드 호출을 감소시켜라

다음과 같은 예처럼 원격 객체의 메소드를 호출할 때

```
remoteObject.getName("name");  
remoteObject.getCity("Austin");  
remoteObject.getState("TX");  
remoteObject.getZipCode("78749");
```

이 예에서 클라이언트로부터 원격 객체에 대한 네 번의 네트워크 호출이 이루어졌다. 이러한 네트워크 호출들은 다음과 같이 메소드 호출의 수를 감소시킴으로써 최소화할 수 있다.

```
// 로컬에서 객체를 생성하고 값을 설정한다.  
PersonInfo person = new PersonInfo();
```

```
person.setName("name");  
person.setCity("Austin");  
person.setState("TX");  
person.setZipCode("78749");  
remoteObject.getPersonInfo(person); // 객체를 네트워크를 통해  
전송한다.
```

이 예에서는 세 번의 네트워크 호출이 단 한번으로 감소했다.

첫 번째 예에서는 속성 값들을 하나씩 설정하였으며, 이들 값들은 네트워크를 통해 전송되었다. 이 방법은 작은 단위(*finer grained*)의 접근 방법이라 불리며 많은 네트워크 호출이 필요하다.

두 번째 예에서는 속성들을 개별적으로 전송하지 않고, 클라이언트 측에서 네 속성 모두를 하나의 객체에 래핑하여 전송하였다. 이 방법은 큰 단위의 (*coarse grained*) 접근 방법이라 불리며 적은 네트워크 호출이 필요하다.

그러므로 퍼포먼스를 향상시키기 위한 방법으로 네트워크 호출의 수를 최소화하기 위해 큰 단위의 (*coarse grained*) 접근 방법을 이용하라.

2.6 원격 EJB의 직렬화를 컨트롤하라

분산/원격 객체를 이용해 애플리케이션을 작성하기로 결정하였다면, 메소드의 어떤 파라미터와 아규먼트를 네트워크를 통해 전송할 것인지를 신중하게 결정해야 한다. 예를 들어 다음과 같이 객체를 파라미터로 넘기는 경우를 생각해 보자.

```
remoteObject.setPersonInfo(person); // 객체를 넘김으로써 원격  
객체를 호출한다.
```

여기서 `PersonInfo` 객체는 직렬화되고 네트워크를 통해 전송될 뿐만 아니라 모든 전체 `PersonInfo` 객체 그래프(임시 변수를 제외한 변수들과 상위 클래스 변수들)가 네트워크를 통해 전송될 것이다. 전체 객체 그래프가 아닌 단지 `PersonInfo` 객체를 전송하려 했는데도 말이다.

이 문제를 해결하기 위해서, 네트워크를 통해 전송될 필요가 없는 속성들에 'transient' 키워드를 사용하라. 더 자세한 정보는 '직렬화' 섹션을 참조하라.

2.7 EJBHome 객체 레퍼런스를 캐싱하라

클라이언트에서 엔터프라이즈 빈을 호출할 때는 먼저 JNDI를 통해 EJBHome 객체를 가져와야 한다. 이는 앞의 그림에서 설명하고 있으며, 이 과정을 위한 코드는 다음과 같다.

```
// InitialContext를 가져온다.  
public Context getInitialContext() throws NamingException{  
    Properties props= new Properties();  
    props.put (Context.PROVIDER_URL,"urlName");  
    props.put(Context.INITIAL_CONTEXT_FACTORY,"nameservice");  
  
    return new InitialContext(props);  
}  
  
// JNDI로부터 홈 레퍼런스를 가져온다.  
public AccountHome getHome () throws NamingException{  
    Context ctx = setInitialContext();  
    Object home=ctx.lookup("JNDIlookupName");  
    return (AccountHome)  
PortableRemoteObject.narrow(home,AccountHome.class);  
}
```

이 코드는 클라이언트가 엔터프라이즈 빈을 호출할 때 매번 실행된다. 이는 비용이 많이 드는 작업이며 낭비이다. 이 문제를 해결하기 위해서 먼저 EJBHome 객체 레퍼런스를 캐싱하고, 매번 JNDI 룩업 과정을 반복하지 않고도 캐시로부터 계속 재사용함으로써 퍼포먼스를 향상시킬 수 있다.

다음의 간단한 코드는 EJBHome 객체 레퍼런스를 캐싱하는 방법을 보여준다.

```
import javax.ejb.*;  
import javax.rmi.*;  
import java.util.*;  
import javax.naming.*;
```

```
public class EJBHomeCache {
// Hashtable에 홈 레퍼런스를 캐싱한다.
    private static Hashtable homes = new Hashtable();
    Context ctx;
    public EJBHomeCache() throws NamingException {
    }

    public static synchronized EJBHome getHome(Class homeClass)
throws NamingException {
        EJBHome home = (EJBHome) this.homes.get(homeClass);
        if(home == null) {
            ctx = getInitialContext();
            home = (EJBHome)
PortableRemoteObject.narrow(ctx.lookup(homeClass.getName()),hom
eClass);
            this.homes.put(homeClass,home);
        }
        return home;
    }

    private Context getInitialContext() throws NamingException {
        Properties props= new Properties();
        props.put (Context.PROVIDER_URL,"urlname");
        props.put(Context.INITIAL_CONTEXT_FACTORY,"nameservice");
        return new InitialContext(props);
    }
} //end class
```

// AccountHome 객체 레퍼런스를 가져오는 클라이언트 코드

```
AccountHome accountHome =
(AccountHome)EJBHomeCache.getHome(AccountHome.Class);
```

이 예에서는 처음에 홈 객체 레퍼런스를 가져오고 이를 캐시에
집어넣는다. 그리고 다음부터는 비용이 비싼 JNDI 룝업 과정 없이
레퍼런스를 재사용한다. 이 기술은 퍼포먼스를 향상시킨다. 이 기술에
사용 가능한 패턴이 존재하며, 자세한 구현은 EJBHomeFactory
패턴(Service Locator 패턴)을 참조하라.

2.8 트랜잭션을 컨트롤하라

EJB에서는 선언적 또는 프로그램적으로 트랜잭션을 사용할 수 있다.

EJB의 선언적인 트랜잭션은 메소드가 시작할 때 트랜잭션이 시작(begin)되고 메소드가 종료될 때 트랜잭션이 종료(commit)되는 것처럼 메소드 레벨에서 이루어진다. 또한 부모 메소드가 서브 메소드를 사용할 경우 트랜잭션은 이들 서브 메소드로 전파된다. 예를 들어, 네 개의 엔티티 빈 메소드를 호출하는 세션 빈 메소드를 작성하였다면, 트랜잭션은 세션 빈 메소드가 시작할 때 시작되며, 메소드가 종료될 때 트랜잭션도 종료된다. 이때 트랜잭션은 네 엔티티 빈 메소드로 전파되며 세션 빈 메소드로 돌아온다. 이는 트랜잭션 전파의 체인과 같이 동작한다.

선언적인 트랜잭션은 Required, RequiredNew, Mandatory, Supports, NotSupported, Never 등과 같은 여섯 개의 트랜잭션 속성을 가지고 있다. 다음과 같이 ejb-jar.xml 디플로이 디스크립터 파일에 각 메소드에 대해 이들 속성들 중 하나를 선언할 수 있다.

```
<ejb-jar>
<session><transaction-type>Container<<transaction-
type></session>
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>EJBName</ejb-name>
      <method-name>methodName / *</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>
```

각 메소드 이름에 대해 개별적으로 속성을 설정할 수 있으며, 아스테리크(*) 문자를 이용하여 모든 메소드에 속성을 설정할 수도 있다.

선언적 트랜잭션의 문제는 트랜잭션을 위한 코드를 작성하는 대신에 디플로이먼트 디스크립터 파일에 트랜잭션 속성들을 선언하기 때문에 트랜잭션이 **innocent** 하다는 것이다. 예를 들어, 아스테리크(*) 문자를 사용한다면 모든 메소드들은 트랜잭션과 연관될 것이다. 왜 비 트랜잭션 메소드에 트랜잭션이 필요한가? 이는 불필요하며, 모든 메소드를 실행할 때마다 더 많은 시간이 소요될 것이다. 따라서 모든 메소드에 불필요한 트랜잭션 전파를 막기 위해서는 트랜잭션을 컨트롤해야 한다. 이를 위해 할 수 있는 일은 빈의 메소드들을 트랜잭션 메소드와 비 트랜잭션 메소드로 구분하고 트랜잭션 메소드에만 트랜잭션 속성을 설정하는 것이다. 그리고 비 트랜잭션 메소드에는 'NotSupported'나 'Never' 속성을 설정함으로써 트랜잭션 전파를 막을 수 있다. 그러나 엔티티 빈의 경우 데이터를 커밋하기 위해서 트랜잭션과 연관될 필요가 있으므로 'NotSupported'나 'Never' 속성은 엔티티 빈에 사용될 수 없으며, 따라서 세션 빈의 비 트랜잭션 메소드에만 사용될 수 있음을 주의하라. 어떤 메소드가 다른 세션 빈을 사용한다면 이 과정에서 트랜잭션 전파의 체인을 컨트롤할 수 있다. 그러나 서브 빈들에 트랜잭션이 필요한지의 여부에 대해 신중해야 한다.

`begin()`, `commit()`, `rollback()` 등과 같은 메소드를 가지고 있는 `javax.transaction.UserTransaction` 인터페이스를 이용하여 트랜잭션 코드를 직접 작성하는 것도 가능하다.

그리고 또 유의할 점은 트랜잭션이 완료될때까지 데이터베이스의 데이터를 록(lock)하며 다른 클라이언트가 이 데이터에 접근할 수 없게 하므로, 트랜잭션 메커니즘은 가능한 최소한의 시간 동안 사용되어야 한다는 것이다.

그러므로 퍼포먼스를 향상시키기 위하여 비 트랜잭션 메소드에 트랜잭션의 적용을 피하고 적은 트랜잭션을 작성함으로써 트랜잭션을 컨트롤하라.

2.9 최적의 트랜잭션 타임아웃을 설정하라

트랜잭션 속성을 선언했다면 트랜잭션이 얼마나 오래 실행되는가? 트랜잭션이 특정 시간 제한 내에 일어나도록 해야 한다면, 벤더 의존적인 디플로이먼트 디스크립터 파일에서 트랜잭션 타임아웃 값을 설정하라. 예를 들어, 웹로직 서버의 `weblogic-ejb-jar.xml` 파일은 다음과 같은 엘리먼트를 가지고 있다.

<transaction-descriptor>

<trans-timeout-seconds>120</trans-timeout-seconds>

</transaction-descriptor>

기타 다른 서버의 경우는 벤더에서 제공하는 문서를 참고하라. 트랜잭션의 최대 타임아웃을 컨트롤하기 위해 이 과정을 통해서 최대 트랜잭션 타임을 설정하라. 하지만 기대되는 값보다 적은 시간을 설정한다면 부정적인 결과가 초래될 수 있으므로 주의하라.

2.10 확장성을 위하여 클러스터링을 이용하라

EJB 애플리케이션을 디플로이하고 후에 클라이언트의 숫자가 증가한다면 어떤 일이 일어날까? 아마도 증가된 클라이언트를 처리하기에 하드웨어 리소스가 부족해 질 것이며 응답 시간이 증가하여 퍼포먼스가 감소될 것이다. 이러한 상황을 타개하기 위해서는 클러스터링을 해야 한다.

클러스터링은 클라이언트에게 서비스하는 서버의 그룹이다. 하지만 클라이언트는 다중 서버의 존재를 인식하지 못하며, 하나의 서버와 통신하고 있다고 생각한다. 일단 애플리케이션을 클러스터링된 환경에 디플로이하면, 클라이언트의 증가에 따라 서버를 증가시킬 수 있다. 클러스터링은 클라이언트의 숫자가 증가할 때 좋은 확장성을 주며, 또한 높은 유용성을 선사한다. 주요 EJB 벤더들 중 일부는 클러스터링을 지원한다. 예를 들어, 세 대의 서버에 클러스터링된 EJB 애플리케이션을 디플로이한 상황에서 첫 번째 서버에 과부하가 걸릴 경우, 클라이언트는 로드 밸런싱 알고리즘에 의해 자동으로 다음 서버에게 라우팅된다. 또한 첫 번째 서버가 다운되는 경우에도 클라이언트는 다른 서버에게 라우팅되며, 이 경우 클라이언트 상태는 다른 서버에게 넘겨져 그 시점부터 처리된다(이를 페일 오버라 한다). 애플리케이션을 하나의 서버에 디플로이했을 때에는 클라이언트 수의 증가에 대한 미래 예측을 할 수 없으며, 따라서 연관된 오버헤드 때문에 클러스터링된 EJB의 퍼포먼스가 감소할 수 있으므로 클러스터링을 사용하지 않는 것이 좋다.

클러스터링된 EJB 애플리케이션을 설정하기 위해서는 벤더 의존적인 디플로이먼트 디스크립터 파일(또는 다른 벤더 의존적인 방법을 통해)을 설정할 필요가 있다. 클러스터링을 설정하면, EJB 컨테이너/서버는 EJB의 클러스터링 인식 타입을 생성한다.

클러스터링에 대한 자세한 사항은 벤더가 제공하는 문서를 참고하라.
클러스터링을 이용하면 높은 확장성과 페일 오버를 얻을 수 있다.

2.11 쓰레드 숫자를 튜닝하라

당신이 현재 사용 중인 EJB 서버는 동시에 실행될 수 있는 오퍼레이션 또는 쓰레드의 숫자를 조정(튜닝)할 수 있는 기능을 가지고 있을 수 있다. 서버가 디폴트로 제공하는 쓰레드 숫자 값이 서버의 용량보다 적다면, 애플리케이션에 대한 클라이언트들의 요청이 큐에 저장될 수 있다. 서버의 리소스와 용량에 따라 쓰레드 카운트 값을 조절함으로써 퍼포먼스를 향상시킬 수 있다. 하지만 쓰레드 카운트 값에 대해 확실하게 알 수 없다면 그냥 디폴트 값으로 두는 것이 나을 것이다. 충분한 리소스 없이 쓰레드 카운트를 증가시킨다면 퍼포먼스는 떨어질 것이다. 더 상세한 정보는 벤더에서 제공하는 문서를 참조하라.

2.12 HttpSession과 상태유지 세션 빈 중 하나를 선택하라

서블릿의 HttpSession 객체와 EJB의 상태유지 세션 빈은 클라이언트의 상태를 유지하는 역할을 수행한다. 이 중 어느 것을 선택하는 것이 좋을까? 두 메커니즘의 장점과 단점을 자세히 살펴보자.

① 상태유지 세션 빈

장점 :

상태유지 세션 빈은 트랜잭션 서비스, 보안 서비스, 라이프 사이클 관리, RMI, 인스턴스 캐싱, 쓰레드 안전 등을 지원한다. 따라서 개발자는 이러한 서비스들을 위한 코드를 작성할 필요가 없다.

웹 기반 클라이언트와 비 웹 기반 클라이언트(예를 들면 스윙과 같은) 양쪽 다에 의해 사용될 수 있다.

하나의 http 요청으로 다수의 오퍼레이션을 수행할 수 있다.

단점 :

위에서 언급한 여러가지 서비스를 지원하므로, 요청을 처리하는데 더 많은 시간이 필요하다.

② HttpSession 객체

장점 :

간단한 자바 객체(아마도 Hashtable)이며 클라이언트 상태를 유지하는데 더 적은 시간과 리소스를 잡아먹는다.

단점 :

상태유지 세션 빈에서 언급했던 사항들을 전혀 지원하지 않는다.

하나의 http 요청으로 다수의 오퍼레이션을 실행할 수 없다.

따라서 애플리케이션의 요구사항에 따라 가장 적합한 메커니즘을 선택해야 한다. 단지 클라이언트 상태 관리를 위한 빈을 원한다면 상태유지 세션 빈 보다는 HttpSession 객체가 더 퍼포먼스가 좋을 것이다.

2.13 EJB 서버를 선택하라

30개 이상의 벤더들이 EJB 컨테이너/서버를 제공하고 있다. 따라서 서버를 선택해야 할 필요가 있을 때는 다음의 이슈들을 주의 깊게 살펴보고 그들 간의 트레이드오프를 고려할 필요가 있다. 다음의 특징들은 EJB 서버를 결정하기 전에 봐둘 필요가 있다.

- ① 클러스터링
- ② 로드 밸런싱
- ③ 인스턴스 풀링과 인스턴스 캐싱
- ④ 수동적인 로딩 (Lazy loading)
- ⑤ EJB 1.1을 위한 패스 바이 레퍼런스
- ⑥ 다양한 락킹(locking) 전략
- ⑦ 커넥션 풀링
- ⑧ CMP의 ejbLoad(), ejbStore() 등과 같은 콜백 메소드 제어

EJB 서버가 이러한 모든 특징들을 제공한다면, 서버의 퍼포먼스를 어떻게 확인할 수 있겠는가? 해답은 TPC-W, @Bench 등과 같이 유용한 벤치마킹 스펙을 사용하여 서버를 테스트하는 것이다. 그러나 이러한 벤치마킹 정책들은 단지 서버의 특정한 특징들만을 테스트한다는 것에 있으며, 따라서 서버의 전반적인 퍼포먼스를 테스트하기 위해서는 SUN에서 릴리즈한 ECperf 스펙을 사용할 수 있다. ECperf 벤치마크 스펙은 EJB 서버의 퍼포먼스 테스트를 위하여 java community process에서 개발되었다. ECperf 스펙은 EJB 서버를

테스트하기 위한 EJB 코드와 드라이버를 제공한다. ECperf 스펙에 대한 자세한 정보는 다음의 링크를 참고하라.

<http://java.sun.com/j2ee/ecperf/>

또한 다음의 사이트에서 다양한 서버들의 벤치마킹 결과를 확인할 수 있다.

<http://ECperf.theserverside.com/ecperf/>

<http://www.cmis.csiro.au/adsat/>

최고의 EJB 서버를 선택하려면 서로 다른 EJB 서버들의 특징들을 분석하라. 그리고 ECperf 툴킷을 이용하여 테스트하고 (위의 링크에 있는 벤치마킹 결과와 같은) 벤치마킹 결과를 확인하라. 마지막으로 애플리케이션에 적합한 서버를 선택하라.

3. 무상태 세션 빈을 튜닝하라.

이미 앞의 섹션에서 여러 공통적인 주제에 대해 살펴보았다. 이러한 주제들은 무상태 세션 빈에도 마찬가지로 적용된다. 여기서는 무상태 세션 빈에 대한 특정한 사항들을 살펴볼 것이다. 이러한 사항들에 대해 자세히 설명하기 위해, 먼저 무상태 세션 빈의 라이프 사이클을 살펴볼 것이다. ...

3.1 무상태 세션 빈 라이프 사이클

라이프 사이클이란 EJB 컨테이너에 의해 엔터프라이즈 빈이 생성되고 제거되며 컨테이너가 엔터프라이즈 빈의 메소드를 호출하는 과정을 의미한다. 다음의 그림은 무상태 세션 빈의 라이프 사이클이다.

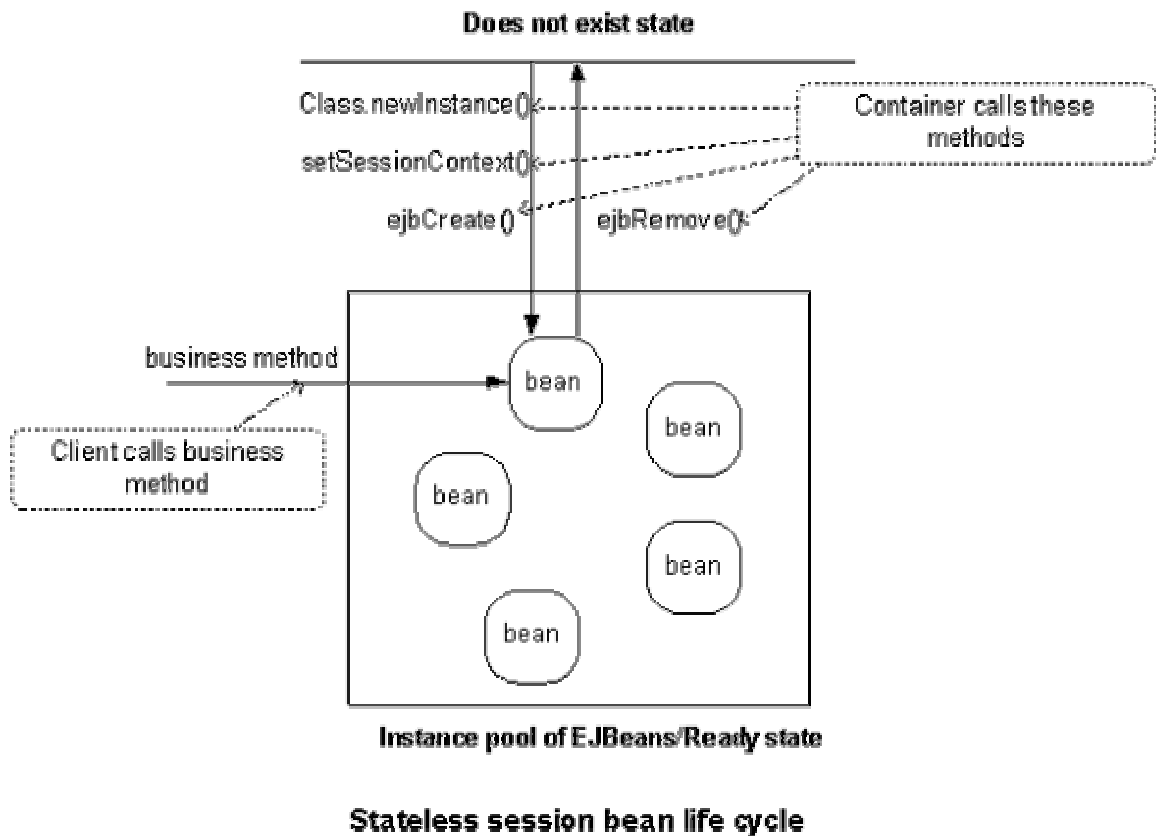


그림 6.

라이프 사이클을 컨트롤 하는 방법은 벤더 종속적인 디플로이먼트 디스크립터에 인스턴스 풀 사이즈를 명시하는 것이다. 예를 들어 웹로직 서버의 `weblogic-ejb-jar.xml`은 다음과 같이 인스턴스 풀 사이즈를 위한 엘리먼트를 가지고 있다.

```
<pool>  
  <max-beans-in-free-pool>100</max-beans-in-free-pool>  
  <initial-beans-in-free-pool>50</initial-beans-in-free-pool>  
</pool>
```

그리고 JBoss 서버의 `jboss.xml`은 풀 사이즈 설정을 위한 `<instance-pool>` 엘리먼트를 가지고 있다. 다른 서버에 대한 자세한 정보는 벤더가 제공하는 문서를 참고하라. 여기서 풀의 초기 빈들과 최대 빈들의 숫자를 기술할 수 있다. 예를 들어 풀의 초기 빈들을 50으로 설정하고 최대 빈들의 개수를 100으로 설정한다면, 서버가 시작될 때 라이프 사이클이 시작된다.

서버가 시작될 때, EJB 컨테이너/서버는 `Class.newInstance()` 메소드를 이용하여 50개의 빈들을 생성하고 풀에 집어넣는다. 그리고 EJB 컨테이너/서버는 다음과 같은 빈의 콜백 메소드를 호출한다.

```
setSessionContext(ctx) and  
ejbCreate() methods
```

이들 50개의 빈들은 50개의 클라이언트들에 의해 동시에 접근될 준비가 되어있다. 동시에 사용되는 빈들이 50개 이상으로 늘어난다면 컨테이너는 빈들이 100개(풀의 최대 빈들의 개수)가 될 때까지 빈들을 생성하고 위의 메소드들을 호출한다. 그리하여 어떤 시점이 되면 컨테이너는 단지 최대 100개의 빈들만을 가지고 클라이언트들에게 서비스할 수 있다. 동시 클라이언트들의 수가 100개 이상이 된다면 어떤 일이 벌어질까? 컨테이너는 클라이언트들을 큐에 집어넣을 것이다. 이 문제에 대해서는 풀 사이즈를 튜닝하는 방법이라는 주제로 다음 섹션에서 다룰 것이다.

컨테이너는 빈에 접근하는 클라이언트의 수가 적어지면 풀로부터 빈을 제거한다. 컨테이너는 일정한 알고리즘(보통 LRU 알고리즘, Least Recently Used)에 의해 빈을 제거한다. 이때 컨테이너는 빈의 `ejbRemove()` 메소드를 호출한다.

클라이언트가 `home.create()` 메소드를 호출하면, 컨테이너는 `EJBObject`를 생성하고 풀에 존재하는 빈을 클라이언트에게 할당한다. 이때 클라이언트는 빈을 생성하지도, `ejbCreate()` 메소드를 호출하지도 않는다. 빈은 이미 생성되어 있고 `ejbCreate()` 메소드 역시 이미 호출되어 있기 때문이다. 마찬가지로 클라이언트가 `home.remove()`나 `remote.remove()` 메소드를 호출하면 컨테이너는 `EJBObject`를 제거하고 클라이언트에게 할당된 빈을 가져와 풀에 다시 집어넣을 뿐 실제로 제거하지 않는다. 무상태 세션 빈의 라이프 사이클에서 클라이언트는 빈의 라이프 사이클이나 빈의 콜백 메소드에 대한 제어를 하지 않는다. 클라이언트는 빈의 생성 및 파괴, 그리고 콜백 메소드의 호출에 대해서 전혀 알지 못한다. 따라서 무상태 세션 빈의 생성 및 파괴, 그리고 콜백 메소드의 호출은 풀의 사이즈, 클라이언트의 동시 접근, 그리고 컨테이너의 알고리즘에 의해서 결정된다.

이름이 암시하듯, 무상태 세션 빈은 메소드 간에 어떠한 상태(인스턴스 변수 값)도 유지하지 않으며, 이것이 바로 무상태 세션 빈의 `ejbActivate()`와 `ejbPassivate()` 메소드가 중요하지 않은 이유이다. 따라서 컨테이너는 각각의 연속적인 메소드의 호출에도 클라이언트에게 풀의 서로 다른 빈을 할당할 수 있다.

이번 섹션에서는 무상태 세션 빈에서 풀 사이즈의 중요성과 콜백 메소드가 실행되는 시점을 이해하였다. 이제 무상태 세션 빈을 튜닝하는 방법을 설명할 것이다.

3.2 무상태 세션 빈의 인스턴스 풀 사이즈를 튜닝하라

빈의 생성과 파괴는 비용이 많이 든다. 이러한 비용을 줄이기 위해서 EJB 컨테이너/서버는 벤더 종속적인 설정에 따라 빈의 풀을 생성하며, 개발자는 퍼포먼스를 증가시키기 위해 이러한 풀 사이즈에 적당한 값을 설정할 필요가 있다. 앞에서 설명했듯이 이러한 풀 사이즈(초기 빈 개수와 최대 빈 개수)는 설정이 가능하다. 예를 들어, 웹로직은 `weblogic-ejb-jar.xml`의 `<pool>` 엘리먼트가, 그리고 JBoss 서버는 `jboss.xml`의 `<instance-pool>` 엘리먼트를 통해 설정할 수 있다. 현재 사용중인 서버의 풀 사이즈 설정에 대해서는 벤더에서 제공하는 문서를 참고하라.

풀에서의 최대 빈 개수는 퍼포먼스에 큰 영향을 미친다. 이 값이 작으면, 빈에 접근하는 클라이언트의 수가 풀의 최대 사이즈보다 클 때 컨테이너는 클라이언트를 큐에 집어넣게 된다. 이는 퍼포먼스를 떨어뜨리고 클라이언트는 실행에 더 많은 시간이 필요하게 된다. 최고의 퍼포먼스를 위해서는 최대 클라이언트 접근의 수와 같게 최대 빈 개수를 설정하라.

3.3 `setSessionContext()` 또는 `ejbCreate()` 메소드를 캐싱하여 사용하라

무상태 세션 빈의 라이프 사이클에서 컨테이너는 먼저 빈 인스턴스를 생성하여 `setSessionContext(SessionContext sc)`와 `ejbCreate()` 메소드를 실행한 후 풀에 집어 넣는다. 그리고 나중에 이 빈들은 컨테이너에 의해 제거될 때까지 다른 클라이언트에 의해 사용된다. 이들 메소드를 이용하면 다른 세션 빈이나 엔티티 빈의 홈 객체 레퍼런스나 `DataSource` 레퍼런스와 같은 리소스를 획득할 수 있으며 이를 인스턴스 변수에 넣을 수 있다. 이러한 리소스를 한번 획득하면, 이미 획득한 리소스를 다른 클라이언트를 위해 사용 가능하므로 각각의 클라이언트를 위해 리소스를 생성할 필요가 없어진다. 분명 이러한 리소스들은 빈에 종속적이며 전역적으로 사용될 수 없다. 전역적인 재사용을 위해서는 앞에서 설명했던 `EJBHome` 객체 레퍼런스를 캐싱하는 것이 더 좋은 방법이다. 또한 이 방법은 다른 리소스를 획득하는데도 사용할 수 있다. 이러한 메소드를 이용하여 데이터베이스 커넥션과 같은 물리적인 리소스를 획득하면 안 된다는 사실을 명심하라. 만약 동시적인 클라이언트가 많아지고 풀 사이즈가 커진다면, 그러한 종류의 리소스는 각각의 메소드내에서 획득하고 제거하는 것이 더 낫다. 다른 클라이언에게도 필요한 빈 종속적인 리소스를 캐싱하는데 `setSessionContext(SessionContext sc)`와 `ejbCreate()` 메소드를 사용하라.

3.4 `ejbRemove()` 메소드에서 리소스를 해제하라.

컨테이너는 풀에서 빈을 제거하기 직전에 `ejbRemove()` 메소드를 호출한다. 따라서 (앞에서 설명한 것과 같이) 빈에서 획득한 리소스들은 이 메소드에서 해제되어야 한다.

4. 상태유지 세션 빈을 튜닝하라

앞에서 설명했던 공통적인 특성들은 상태유지 세션 빈에도 마찬가지로 적용된다. 여기서는 상태유지 세션 빈의 특징적인 특성에 대해서 설명할 것이다. 먼저 상태유지 세션 빈의 라이프 사이클을 살펴보자.

4.1 상태유지 세션 빈의 라이프 사이클

상태유지 세션 빈과 무상태 세션 빈의 라이프 사이클을 서로 다르다. 그 이유는 상태유지 세션 빈의 경우 클라이언트의 메소드 호출 사이에 상태(인스턴스 변수 값)를 유지할 필요가 있기 때문이다. 이는 클라이언트가 메소드를 이용하여 인스턴스 변수의 값을 생성하면, 그 값은 다음 메소드 호출에서도 사용 가능하다는 것을 의미한다. 다음의 그림은 상태유지 세션 빈의 라이프 사이클이다.

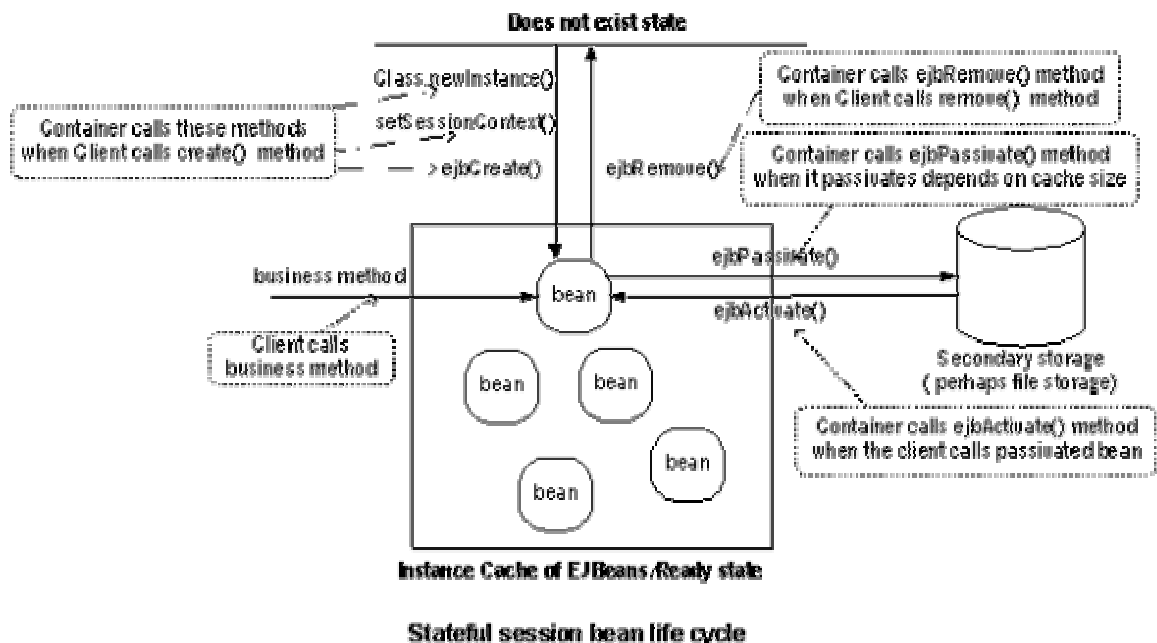


그림 7 : 상태유지 세션 빈 라이프 사이클

여기서 인스턴스 풀이 아닌 인스턴스 캐시를 볼 수 있다. 풀이 상태를 가지지 않은 (무상태 Stateless) 빈을 유지하는 반면, 캐시는 상태를 가진 (상태유지 Stateful) 빈을 유지한다. 라이프 사이클을 컨트롤하는 방법은 벤더 종속적인 디플로이먼트 디스크립터 파일에 인스턴스 캐시의 사이즈를 기술하는 것이다. 예를 들어, 웹로직의 `weblogic-ejb-jar.xml`에서는 다음과 같은 엘리먼트를 통해 인스턴스 캐시 사이즈를 설정한다.

```
<stateful-session-cache>  
    <max-beans-in-cache>100</max-beans-in-cache>  
</stateful-session-cache>
```

JBoss 서버의 `jboss.xml`은 다음과 같은 엘리먼트를 통해 인스턴스 캐시 사이즈를 설정한다.

```
<instance-cache>  
<container-cache-conf>  
    <cache-policy>  
        <cache-policy-conf>  
            <min-capacity>5</min-capacity>  
            <max-capacity>10</max-capacity>  
        </cache-policy-conf>  
    </cache-policy>  
</container-cache-conf>  
</instance-cache>
```

인스턴스 캐시 설정에 대한 더 자세한 정보는 벤더에서 제공하는 문서를 참고하라.

여기서 개발자는 캐시의 최소 빈 개수와 최대 빈 개수를 기술할 수 있다. 웹로직과 같은 몇몇 벤더들은 최소 빈 개수의 설정을 지원하지 않지만 최대 빈 개수의 설정은 지원한다. 현재 사용중인 EJB 컨테이너/서버가 이러한 설정을 지원하는지의 여부는 벤더에서 제공하는 문서를 참고하라. 서버의 벤더에 따라 캐시의 최소 빈 개수와 최대 빈 개수를 설정할 수 있다.

무상태 세션 빈과는 달리 상태유지 세션 빈의 라이프 사이클은 클라이언트가 `create()` 메소드를 호출함으로써 시작된다. 클라이언트가 `create()` 메소드를 호출할 때, 컨테이너는 `Class.newInstance()`를 이용하여 빈을 생성하고 다음의 메소드를 호출한다.

```
setSessionContext(ctx)
ejbCreate()
```

그리고 생성된 빈을 인스턴스 캐시에 집어 넣는다. 그리고 나서 클라이언트는 `remove()` 메소드를 호출해 컨테이너가 `ejbRemove()` 메소드를 호출하고 빈을 파괴할 때까지 해당 빈을 할당 받는다. 상태유지 세션 빈에서는 클라이언트가 라이프 사이클(활성화와 비활성화를 제외한 생성과 파괴)을 컨트롤한다. 컨테이너는 언제 `ejbActivate()`와 `ejbPassivate()` 메소드를 호출하는가? 예를 들어, 캐시의 최대 빈 개수를 100으로 설정한다면 클라이언트가 동시에 빈에 접근할 때 컨테이너는 100개(최대 빈 개수)까지 빈을 생성하고 이들 빈을 클라이언트에게 할당한다. 이후 101번째 클라이언트가 이 빈에 접근하면 컨테이너는 알고리즘(보통 LRU 알고리즘, Least Recently Used)에 의해 몇몇 idle한 빈을 비활성화시킨다. 빈을 비활성화시키기 직전에 컨테이너는 `ejbPassivate()` 메소드를 호출한다. 여기서 비활성화란 빈의 상태(인스턴스 변수 값)를 2차 저장소(아마도 파일 저장소)에 저장하는 것을 의미한다. 비활성화는 직렬화를 통해 일어난다. 나중에 idle 클라이언트가 다시 빈에 접근하면, 컨테이너는 빈을 활성화시키고 활성화된 값을 인스턴스 값에 재할당한 후 `ejbActivate()` 메소드를 호출한다.

여기서 이해해야 할 것은 클라이언트가 빈의 생성과 파괴를 컨트롤하지만 활성화와 비활성화는 컨트롤하지 않는다는 것이다. 활성화와 비활성화는 인스턴스 캐시 사이즈에 따라 컨테이너에 의해서 컨트롤된다.

이번 섹션에서는 상태유지 세션 빈에서의 인스턴스 캐시 사이즈의 중요성과 언제 콜백 메소드가 실행 되는지에 대해서 설명하였다. 이제 상태유지 세션 빈을 튜닝하는 방법에 대해 알아보자.

4.2 상태유지 세션 빈의 인스턴스 캐시 사이즈를 튜닝하라

앞에서 설명했듯이 개발자는 벤더 종속적인 디플로이먼트 디스크립터 파일에 인스턴스 캐시 사이즈를 기술함으로써 간접적으로 활성화와 비활성화를 제어할 수 있다. 활성화와 비활성화는 직렬화 과정 때문에 비용이 매우 비싸다. 인스턴스 캐시 사이즈가 작고 동시에 활성화된 클라이언트가 인스턴스 캐시 사이즈보다 많다면 활성화와 비활성화 과정이 자주 일어나며, 따라서 퍼포먼스가 떨어진다. 그러므로 퍼포먼스를 향상시키기 위해서는 빈에 접근하는 동시 활성화 클라이언트의 수와 같은 최적화된 캐시 사이즈를 설정해야 한다.

4.3 상태유지 세션 빈을 위해 최적화된 빈 연령을 설정하라

컨테이너에 의한 인스턴스 캐시로부터의 엔터프라이즈 빈 제거는 클라이언트의 `remove()` 메소드 호출뿐만 아니라 벤더 종속적인 디스크립터 파일에 설정된 엔터프라이즈 빈 타임아웃 값(빈 연령)에 의해서도 일어난다. 빈 타임아웃 값이 작다면 컨테이너는 빈을 자주 제거하고 다시 생성해야 하며, 이 과정은 매우 비용이 비싸다. 따라서 잦은 제거 및 생성 과정을 최소화하기 위해서는 최적화된 빈 연령을 설정하라.

4.4 상태유지 세션 빈의 직렬화를 컨트롤하라

이미 앞에서 네트워크를 통해 클라이언트로부터 엔터프라이즈 빈에게 메소드 파라미터를 넘길 때 직렬화를 컨트롤하는 방법을 설명함으로써 원격 EJB의 직렬화 제어에 대해서 다루었다. 여기서는 상태유지 세션 빈의 직렬화에 대해 설명할 것이다.

상태유지 세션 빈이 비활성화될 때 컨테이너는 인스턴스 변수 값들을 직렬화하고 그 결과를 2차 저장소(파일 시스템이나 데이터베이스)에 저장해야 한다. 그리고 빈이 다시 활성화될 때 그 값들을 역직렬화해야 한다. 직렬화는 'transient' 변수를 제외한 모든 인스턴스 변수들에게서 일어난다. 인스턴스 변수가 거대한 객체 그래프를 가지고 있다면, 전체 그래프나 모든 인스턴스 변수들을 직렬화하는 것을 원하지 않을 수도 있다. 따라서 원하지 않는 변수들에 대한 불필요한 직렬화를 피하기 위하여 그러한 변수들에 'transient'

키워드를 사용함으로써 직렬화 과정이 감소될 수 있다. 직렬화에 대한 더 자세한 정보는 직렬화 섹션을 참고하라.

4.5 상태유지 세션 빈을 명시적으로 제거하라

컨테이너는 빈의 타임아웃이 일어나거나 클라이언트가 `remove()` 메소드를 명시적으로 호출하여 인스턴스 캐시에 빈을 유지한다. 따라서 클라이언트가 빈에 대한 작업을 종료하고도 명시적으로 빈을 제거하지 않는다면, 컨테이너는 빈을 인스턴스 캐시에 유지하거나 빈을 비활성화 시킨다. 이 과정은 불필요한 메모리를 낭비하게 된다. 따라서 `remove()` 메소드를 이용하여 클라이언트로부터 빈을 명시적으로 제거하라.