- **2차 캐시**
- **1차 캐시와 2차 캐시**
  - **- 1차 캐시**
  - **- 2차 캐시**
- **JPA 2차 캐시 기능**
  - **- 캐시 모드 설정**
  - **- 캐시 조회, 저장 방식 설정**
  - **- JPA 캐시 관리 API**
- **하이버네이트와 EHCACHE 적용하기**
  - **- 환경 설정**
  - **- 엔티티 캐시와 컬렉션 캐시**
  - **- @Cache**
  - **- 캐시 영역**
  - **- 쿼리 캐시**
  - **- 쿼리 캐시 영역**
  - **- 쿼리 캐시와 컬렉션 캐시의 주의점**

2차 캐시

JPA가 제공하는 애플리케이션 범위의 캐시에 대해 알아보고 하이버네이트와 EHCACHE를 사용해서 실제 캐시를 적용해보자.

1차 캐시와 2차 캐시

네트워크를 통해 데이터베이스에 접근하는 시간 비용은 애플리케이션 서버에서 내부 메모리에 접근하는 시간 비용보다 수만에서 수십만 배 이상 비싸다. 따라서 조회한 데이터를 메모리에 캐시 해서 데이터베이스 접근 횟수를 줄이면 애플리케이션 성능을 획기적으로 개선할 수 있다.

영속성 컨텍스트 내부에는 엔티티를 보관하는 저장소가 있는데 이것을 1차 캐시라 한다. 이것으로 얻을 수 있는 이점이 많지만, 일반적인 웹 애플리케이션 환경은 트랜잭션을 시작하고 종료할 때까지만 1차 캐시가 유효하다. OSIV를 사용해도 클라이언트의 요청이 들어올 때부터 끝날 때까지만 1차 캐시가 유효하다. 따라서 애플리케이션 전체로 보면 데이터베이스 접근 횟수를 획기적으로 줄이지는 못한다.

하이버네이트를 포함한 대부분의 JPA 구현체들은 애플리케이션 범위의 캐시를 지원하는데 이것을 공유 캐시 또는 2차 캐시라 한다. 이런 2차 캐시를 활용하면 애플리케이션 조회 성능을 향상할 수 있다.

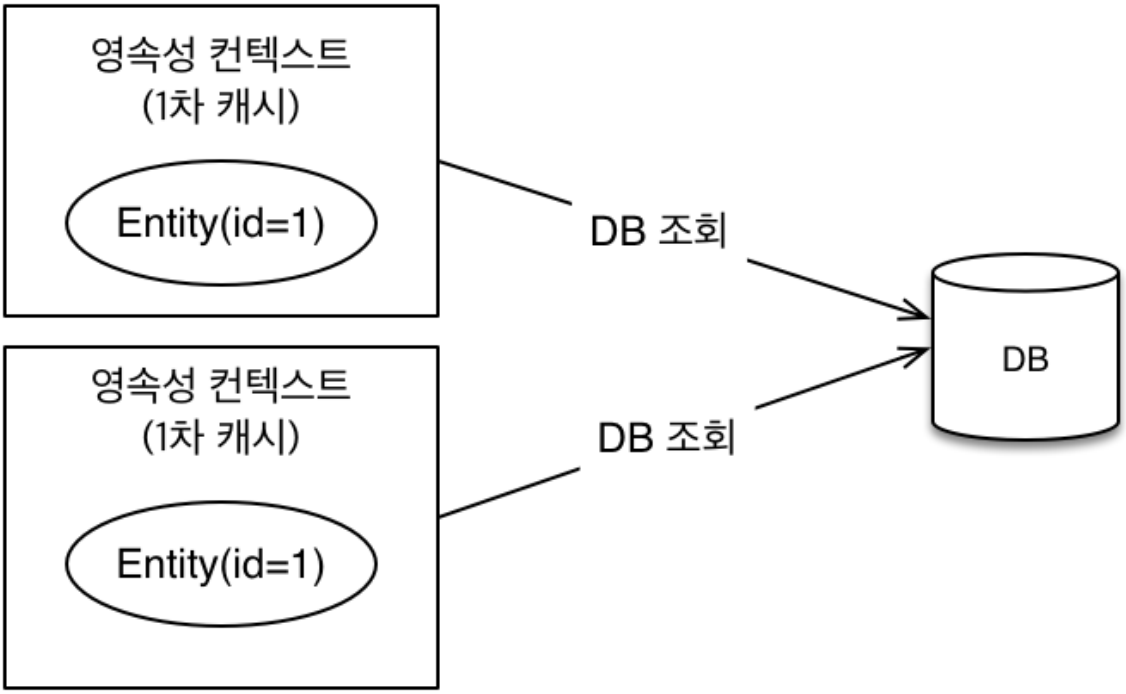

그림 - 2차 캐시 적용 전

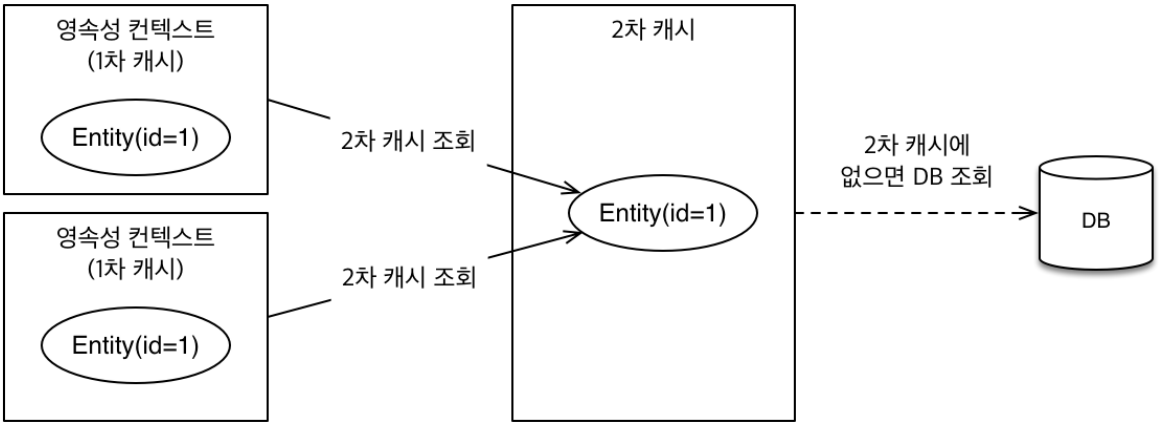

그림 - 2차 캐시 적용 후

- 1차 캐시

1차 캐시는 영속성 컨텍스트 내부에 있다. 엔티티 매니저로 조회하거나 변경하는 모든 엔티티는 1차 캐시에 저장된다. 트랜잭션을 커밋하거나 플러시를 호출하면 1차 캐시에 있는 엔티티의 변경 내역을 데이터베이스에 동기화 한다.

JPA를 J2EE나 스프링 프레임워크 같은 컨테이너 위에서 실행하면 트랜잭션을 시작할 때 영속성 컨텍스트를 생성하고 트랜잭션을 종료할 때 영속성 컨텍스트도 종료한다. 물론 OSIV를 사용하면 요청(예를 들어 HTTP 요청)의 시작부터 끝까지 같은 영속성 컨텍스트를 유지한다.

1차 캐시는 끄고 켤 수 있는 옵션이 아니다. 영속성 컨텍스트 자체가 사실상 1차 캐시다.

1차 캐시 동작 방식

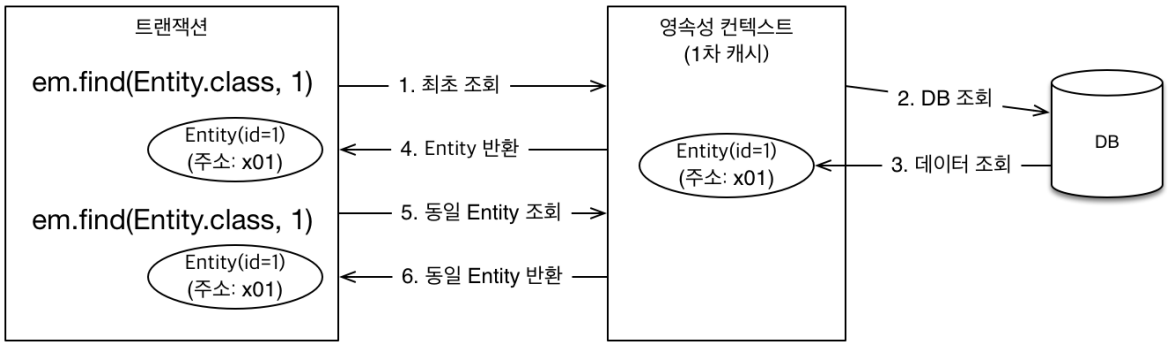

그림 - 1차 캐시 동작 방식

[그림 - 1차 캐시 동작 방식]을 분석해보자.

- (1) 최초 조회할 때는 1차 캐시에 엔티티가 없으므로
- (2) 데이터베이스에서 엔티티를 조회해서
- (3) 1차 캐시에 보관하고
- (4) 1차 캐시에 보관한 결과를 반환한다.
- (5) 이후 같은 엔티티를 조회하면 1차 캐시에 같은 엔티티가 있으므로 데이터베이스를 조회하지 않고 1차 캐시의 엔티티를 그대로 반환한다.

1차 캐시의 특징은 다음과 같다.

- 1차 캐시는 같은 엔티티가 있으면 해당 엔티티를 그대로 반환한다. 따라서 1차 캐시는 객체 동일성(a == b)을 보장한다.
- 1차 캐시는 기본적으로 영속성 컨텍스트 범위의 캐시다. (컨테이너 환경에서는 트랜잭션 범위의 캐시, OSIV를 적용하면 요청 범위의 캐시다.)

- 2차 캐시

애플리케이션에서 공유하는 캐시를 JPA는 공유 캐시(shared cache)라 하는데 일반적으로 2차 캐시(second level cache, L2 cache)라 부른다. 2차 캐시는 애플리케이션 범위의 캐시다. 따라서 애플리케이션을 종료할 때까지 캐시가 유지된다. 분산 캐시나 클러스터링 환경의 캐시는 애플리케이션보다 더 오래 유지될 수도 있다. 2차 캐시를 적용하면 엔티티 매니저를 통해 데이터를 조회할 때 우선 2차 캐시에서 찾고 없으면 데이터베이스에서 찾는다. 2차 캐시를 적절히 활용하면 데이터베이스 조회 횟수를 획기적으로 줄일 수 있다.

2차 캐시 동작 방식

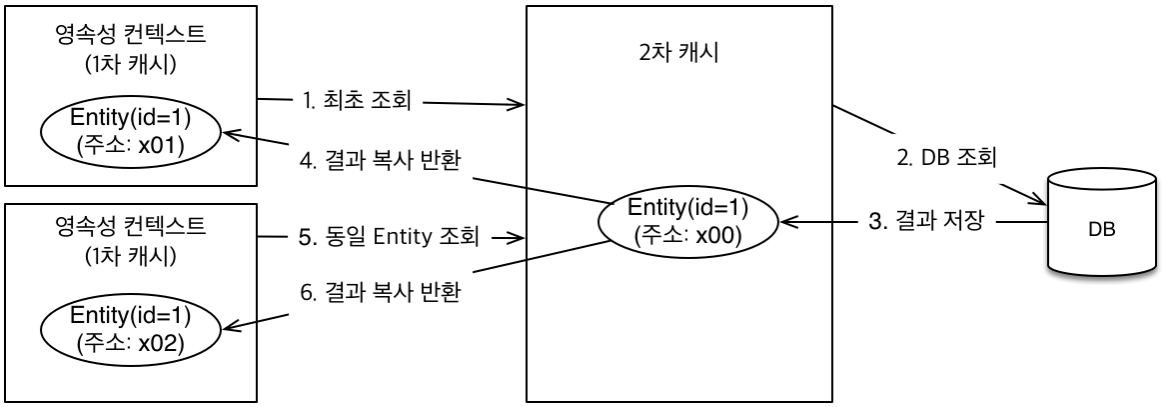

그림 - 2차 캐시 동작 방식

[그림 - 2차 캐시 동작 방식]을 분석해보자.

- (1) 영속성 컨텍스트는 엔티티가 필요하면 2차 캐시를 조회한다.
- (2) 2차 캐시에 엔티티가 없으면 데이터베이스를 조회해서
- (3) 결과를 2차 캐시에 보관한다.
- (4) 2차 캐시는 자신이 보관하고 있는 엔티티를 복사해서 반환한다.
- (5) 2차 캐시에 저장되어 있는 엔티티를 조회하면 복사본을 만들어 반환한다.

2차 캐시는 동시성을 극대화하려고 캐시한 객체를 직접 반환하지 않고 복사본을 만들어서 반환한다. 만약 캐시한 객체를 그대로 반환하면 여러 곳에서 같은 객체를 동시에 수정하는 문제가 발생할 수 있다. 이 문제를 해결하려면 객체에 락을 걸어야 하는데 이렇게 하면 동시성이 떨어질 수 있다. 락에 비하면 객체를 복사하는 비용은 아주 저렴하다. 따라서 2차 캐시는 원본 대신에 복사본을 반환한다.

2차 캐시의 특징은 다음과 같다.

- 2차 캐시는 영속성 유닛 범위의 캐시다.
- 2차 캐시는 조회한 객체를 그대로 반환하는 것이 아니라 복사본을 만들어서 반환한다.
- 2차 캐시는 데이터베이스 기본 키를 기준으로 캐시하지만 영속성 컨텍스트가 다르면 객체 동일성(a

== b)을 보장하지 않는다.

JPA 2차 캐시 기능

지금부터 캐시라 하면 2차 캐시로 이해하자. 1차 캐시는 명확하게 1차 캐시라 하겠다.

JPA 구현체 대부분은 캐시 기능을 각자 지원했는데 JPA는 2.0에 와서야 캐시 표준을 정의했다. JPA 캐시 표준은 여러 구현체가 공통으로 사용하는 부분만 표준화해서 세밀한 설정을 하려면 구현체에 의존적인 기능을 사용해야 한다.

JPA 캐시 표준 기능을 알아보자.

- 캐시 모드 설정

2차 캐시를 사용하려면 엔티티에 `javax.persistence.Cacheable` 어노테이션을 사용하면 된다. `@Cacheable`은 `@Cacheable(true)`, `@Cacheable(false)`를 설정할 수 있는데 기본값은 `true`다.

```
@Cacheable //**
@Entity
public class Member {

    @Id @GeneratedValue
    private Long id;
    ...
}
```

다음으로 `persistence.xml`에 `shard-cache-mode`를 설정해서 애플리케이션 전체에(정확히는 영속성 유닛 단위) 캐시를 어떻게 적용할지 옵션을 설정해야 한다.

====== persistence.xml ======

```
<persistence-unit name="test">
    <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
</persistence-unit>
```

스프링 프레임워크를 사용할 때 설정하는 방법은 다음과 같다.

====== XML 설정 ======

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContaine
    <property name="sharedCacheMode" value="ENABLE_SELECTIVE"/>
    ...
```

캐시 모드는 `javax.persistence.SharedCacheMode`에 정의되어 있다. 보통 `ENABLE_SELECTIVE`를 사용한다.

표 - `SharedCacheMode` 캐시 모드 설정

캐시 모드	설명
ALL	모든 엔티티를 캐시한다.
NONE	캐시를 사용하지 않는다.
ENABLE_SELECTIVE	**`Cacheable(true)`로 설정된 엔티티만 캐시를 적용한다.**
DISABLE_SELECTIVE	모든 엔티티를 캐시 하는데 `Cacheable(false)`로 명시된 엔티티는 캐시 하지 않는다.
UNSPECIFIED	JPA 구현체가 정의한 설정을 따른다.

- 캐시 조회, 저장 방식 설정

캐시를 무시하고 데이터베이스를 직접 조회하거나 캐시를 갱신하려면 캐시 조회 모드와 캐시 보관 모드를 사용하면 된다.

```
em.setProperty("javax.persistence.cache.retrieveMode", CacheRetrieveMode.BYPASS)
```

캐시 조회 모드나 보관 모드에 따라 사용할 프로퍼티와 옵션이 다르다.

프로퍼티 이름

- `javax.persistence.cache.retrieveMode` 캐시 조회 모드 프로퍼티 이름
- `javax.persistence.cache.storeMode` 캐시 보관 모드 프로퍼티 이름

옵션

- `javax.persistence.CacheRetrieveMode` 캐시 조회 모드 설정 옵션
- `javax.persistence.CacheStoreMode` 캐시 보관 모드 설정 옵션

캐시 조회 모드

```
public enum CacheRetrieveMode {
    USE,
    BYPASS
}
```

- `USE` : 캐시에서 조회한다. 기본값이다.
- `BYPASS` : 캐시를 무시하고 데이터베이스에 직접 접근한다.

캐시 보관 모드

```
public enum CacheStoreMode {
    USE,
    BYPASS,
    REFRESH
}
```

- `USE` : 조회한 데이터를 캐시에 저장한다. 조회한 데이터가 이미 캐시에 있으면 캐시 데이터를 최신 상태로 갱신하지는 않는다. 트랜잭션을 커밋하면 등록 수정한 엔티티도 캐시에 저장한다. 기본값이다.
- `BYPASS` : 캐시에 저장하지 않는다.
- `REFRESH` : `USE` 전략에 추가로 데이터베이스에서 조회한 엔티티를 최신 상태로 다시 캐시한다.

캐시 모드는 `EntityManager.setProperty()` 로 엔티티 매니저 단위로 설정하거나 더 세밀하게 `EntityManager.find()` , `EntityManager.refresh()` , 그리고 `Query.setHint()` (`TypeQuery` 포함)에 사용할 수 있다.

====== 엔티티 매니저 범위 ======

```
em.setProperty("javax.persistence.cache.retrieveMode", CacheRetrieveMode.BYPASS)
em.setProperty("javax.persistence.cache.storeMode", CacheStoreMode.BYPASS);
```

====== find() ======

```
Map<String, Object> param = new HashMap<String, Object>();
param.put("javax.persistence.cache.retrieveMode", CacheRetrieveMode.BYPASS);
param.put("javax.persistence.cache.storeMode", CacheStoreMode.BYPASS);
```

```
em.find(TestEntity.class, id, param);
```

====== Query ======

```
em.createQuery("select e from TestEntity e where e.id = :id", TestEntity.class)
    .setParameter("id", id)
    .setHint("javax.persistence.cache.retrieveMode", CacheRetrieveMode.BYPASS)
    .setHint("javax.persistence.cache.storeMode", CacheStoreMode.BYPASS)
    .getSingleResult();
```

- JPA 캐시 관리 API

JPA는 캐시를 관리하기 위한 `javax.persistence.Cache` 인터페이스를 제공한다. 이것은 `EntityManagerFactory`에서 구할 수 있다.

====== Cache 관리 객체 조회 ======

```
Cache cache = emf.getCache();
boolean contains = cache.contains(TestEntity.class, testEntity.getId());
System.out.println("contains = " + contains);
```

====== Cache 인터페이스 ======

```
public interface Cache {

    // 해당 엔티티가 캐시에 있는지 여부 확인
    public boolean contains(Class cls, Object primaryKey);

    // 해당 엔티티중 특정 식별자를 가진 엔티티를 캐시에서 제거
    public void evict(Class cls, Object primaryKey);

    // 해당 엔티티 전체를 캐시에서 제거
    public void evict(Class cls);

    // 모든 캐시 데이터 제거
    public void evictAll();

    // JPA Cache 구현체 조회
    public <T> T unwrap(Class<T> cls);
```

```
}
```

JPA가 표준화한 캐시 기능은 여기까지다. 실제 캐시를 적용하려면 구현체의 설명서를 읽어보아야 한다. 하이버네이트와 EHCACHE를 사용해서 실제 2차 캐시를 적용해보자.

하이버네이트와 EHCACHE 적용하기

하이버네이트와 EHCACHE[1]를 사용해서 2차 캐시를 적용해보자.

하이버네이트가 지원하는 캐시는 크게 3가지가 있다.

1. **엔티티 캐시**: 엔티티 단위로 캐시한다. 식별자로 엔티티를 조회하거나 컬렉션이 아닌 연관된 엔티티를 로딩할 때 사용한다.
2. **컬렉션 캐시**: 엔티티와 연관된 컬렉션을 캐시한다. **컬렉션이 엔티티를 담고 있으면 식별자 값만 캐시한다.** (하이버네이트 기능)
3. **쿼리 캐시**: 쿼리와 파리미터 정보를 키로 사용해서 캐시한다. **결과가 엔티티면 식별자 값만 캐시한다.** (하이버네이트 기능)

참고로 JPA 표준에는 엔티티 캐시만 정의되어 있다.

- 환경 설정

하이버네이트에서 EHCACHE를 사용하려면 `hibernate-ehcache` 라이브러리를 `pom.xml`에 추가하자.

====== `pom.xml`에 `hibernate-ehcache` 추가 ======

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>4.3.8.Final</version>
</dependency>
```

`hibernate-ehcache`를 추가하면 `net.sf.ehcache-core` 라이브러리도 추가된다.

EHCACHE는 `ehcache.xml`을 설정파일로 사용한다. 이 파일을 `src/main/resources`에 두자. (클래스패스 루트)

====== `ehcache.xml` 추가 ======

```
<ehcache>
    <defaultCache
            maxElementsInMemory="10000"
            eternal="false"
            timeToIdleSeconds="1200"
            timeToLiveSeconds="1200"
            diskExpiryThreadIntervalSeconds="1200"
            memoryStoreEvictionPolicy="LRU"
            />

</ehcache>
```

다음으로 하이버네이트에 캐시 사용정보를 설정해야 한다. `persistence.xml`에 캐시 정보를 추가하자.

====== `persistence.xml` ======

```
<persistence-unit name="test">
    <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
    <properties>
        <property name="hibernate.cache.use_second_level_cache" value="true"/>
        <property name="hibernate.cache.use_query_cache" value="true"/>
        <property name="hibernate.cache.region.factory_class"
                  value="org.hibernate.cache.ehcache.EhCacheRegionFactory" />
        <property name="hibernate.generate_statistics" value="true"/>
    </properties>
    ...
</persistence-unit>
```

설정한 속성 정보는 다음과 같다.

- `hibernate.cache.use_second_level_cache` : 2차 캐시를 활성화 한다. 엔티티 캐시와 컬렉션 캐시를 사용할 수 있다.
- `hibernate.cache.use_query_cache` : 쿼리 캐시를 활성화 한다.
- `hibernate.cache.region.factory_class` : 2차 캐시를 처리할 클래스를 지정한다. 여기서는 EHCACHE를 사용하므로 `org.hibernate.cache.ehcache.EhCacheRegionFactory`를 적용한다.
- `hibernate.generate_statistics` : 이 속성을 `true`로 설정하면 하이버네이트가 여러 통계정보를 출력해주는데 캐시 적용 여부를 확인할 수 있다. (성능에 영향을 주므로 개발 환경에서만 적용

하는 것이 좋다.)

2차 캐시를 사용할 준비를 완료했다. 이제 캐시를 사용해보자.

- 엔티티 캐시와 컬렉션 캐시

먼저 엔티티 캐시와 컬렉션 캐시를 알아보자.

====== 캐시 적용 코드 ======

```
import javax.persistence.Cacheable
import org.hibernate.annotations.Cache

@Cacheable //** (1)
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE) //** (2)
@Entity
public class ParentMember {

    @Id @GeneratedValue
    private Long id;

    private String name;

    @Cache(usage = CacheConcurrencyStrategy.READ_WRITE) //** (3)
    @OneToMany(mappedBy = "parentMember", cascade = CascadeType.ALL)
    private List<ChildMember> childMembers = new ArrayList<ChildMember>();
    ...
}
```

- `javax.persistence.Cacheable` : 엔티티를 캐시하려면 (1)번처럼 이 어노테이션을 적용하면 된다.
- `org.hibernate.annotations.Cache` : 이 어노테이션은 하이버네이트 전용이다. (2)번처럼 캐시와 관련된 더 세밀한 설정을 할 때 사용한다. 또한 (3)번처럼 컬렉션 캐시를 적용할 때도 사용한다.

여기서 `ParentMember`는 엔티티 캐시가 적용되고 `ParentMember.childMembers`는 컬렉션 캐시가 적용된다.

- @Cache

하이버네이트 전용인 `org.hibernate.annotations.Cache` 어노테이션을 사용하면 세밀한 캐시 설정이 가능하다.

표 - 하이버네이트 `@Cache` 속성

속성	설명
`usage`	`CacheConcurrencyStrategy`를 사용해서 캐시 동시성 전략을 설정한다.
`region`	캐시 지역 설정
`include`	연관 객체를 캐시에 포함할지 선택한다. `all`, `non-lazy` 옵션을 선택할 수 있다. 기본값 `all`

중요한 것은 캐시 동시성 전략을 설정할 수 있는 `usage` 속성이다.
`org.hibernate.annotations.CacheConcurrencyStrategy`를 살펴보자.

표 - `CacheConcurrencyStrategy` 속성

속성	설명
`NONE`	캐시를 설정하지 않는다.
`READ_ONLY`	읽기 전용으로 설정한다. 등록, 삭제는 가능하지만 수정은 불가능하다. 참고로 읽기 전용인 불변 객체는 수정되지 않으므로 하이버네이트는 2차 캐시를 조회할 때 객체를 복사하지 않고 원본 객체를 반환한다.
`NONSTRICT_READ_WRITE`	엄격하지 않은 읽고 쓰기 전략이다. 동시에 같은 엔티티를 수정하면 데이터 일관성이 깨어질 수 있다. EHCACHE는 데이터를 수정하면 캐시 데이터를 무효화 한다.
`READ_WRITE`	읽기 쓰기가 가능하고 READ COMMITTED 정도의 격리 수준을 보장한다. EHCACHE는 데이터를 수정하면 캐시 데이터도 같이 수정한다.
`TRANSACTIONAL`	컨테이너 관리 환경에서 사용할 수 있다. 설정에 따라 REPEATABLE READ 정도의 격리 수준을 보장받을 수 있다.

캐시 종류에 따른 동시성 전략 지원 여부는 하이버네이트 공식 문서[2]가 제공하는 다음 표를 참고하자.
여기서 `ConcurrentHashMap`은 개발시에만 사용해야 한다.

====== 표 - 캐시 동시성 전략 지원 여부 ======

Cache	read-only	nonstrict-read-write	read-write	transactional
ConcurrentHashMap	yes	yes	yes	
EHCache	yes	yes	yes	yes
Infinispan	yes			yes

- 캐시 영역

위에서 캐시를 적용한 코드는 다음 캐시 영역(Cache Region)에 저장된다.

- 엔티티 캐시 영역: `jpabook.jpashop.domain.test.cache.ParentMember`
- 컬렉션 캐시 영역: `jpabook.jpashop.domain.test.cache.ParentMember.childMembers`

엔티티 캐시 영역은 기본값으로 [패키지 명 + 클래스 명]을 사용하고, 컬렉션 캐시 영역은 엔티티 캐시 영역 이름에 캐시한 컬렉션의 필드명이 추가된다. 필요하다면 `@Cache(region = "customRegion", ...)` 처럼 캐시 영역을 직접 지정할 수 있다.

캐시 영역을 위한 접두사를 설정하려면 `persistence.xml` 설정에 `hibernate.cache.region_prefix`를 사용하면 된다. 예를 들어 `core`로 설정하면 `core.jpabook.jpashop...`으로 설정된다.

캐시 영역이 정해져 있으므로 영역별로 세부 설정을 할 수 있다. 만약 `ParentMember`를 600초 마다 캐시에서 제거하고 싶으면 EHCACHE를 다음과 같이 설정하면 된다.

====== EHCACHE 세부 설정(`ehcache.xml`) ======

```
<ehcache>
    <defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="1200"
        timeToLiveSeconds="1200"
        diskExpiryThreadIntervalSeconds="1200"
        memoryStoreEvictionPolicy="LRU" />
    <cache
        name="jpabook.jpashop.domain.test.cache.ParentMember"
        maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="600"
```

```
        timeToLiveSeconds="600"
        overflowToDisk="false" />
</ehcache>
```

EHCACHE[3]에 대한 자세한 내용은 공식 문서를 확인하자.

- 쿼리 캐시

쿼리 캐시는 쿼리와 파리미터 정보를 키로 사용해서 쿼리 결과를 캐시하는 방법이다.

쿼리 캐시를 적용하려면 영속성 유닛을 설정에 `hibernate.cache.use_query_cache` 옵션을 꼭 `true`로 설정해야 한다. 그리고 쿼리 캐시를 적용하려는 쿼리마다 `org.hibernate.cacheable`을 `true`로 설정하는 힌트를 주면 된다.

====== 쿼리 캐시 적용 ======

```
em.createQuery("select i from Item i", Item.class)
    .setHint("org.hibernate.cacheable", true) //**
    .getResultList();
```

====== NamedQuery에 쿼리 캐시 적용 ======

```
@Entity
@NamedQuery(
        hints = @QueryHint(name = "org.hibernate.cacheable", value = "true"), //
        name = "Member.findByUsername",
        query = "select m.address from Member m where m.name = :username"
)
public class Member {
    ...
```

- 쿼리 캐시 영역

`hibernate.cache.use_query_cache` 옵션을 `true`로 설정해서 쿼리 캐시를 활성화하면 다음 두 캐시 영역이 추가된다.

- `org.hibernate.cache.internal.StandardQueryCache` : 쿼리 캐시를 저장하는 영역이다. 이곳에는 쿼리, 쿼리 결과 집합, 쿼리를 실행한 시점의 타임스탬프를 보관한다.
- `org.hibernate.cache.spi.UpdateTimestampsCache` : 쿼리 캐시가 유효한지 확인하기 위해

> 쿼리 대상 테이블의 가장 최근 변경(등록, 수정, 삭제) 시간을 저장하는 영역이다. 이곳에는 테이블명과 해당 테이블의 최근 변경된 타임스탬프를 보관한다.

쿼리 캐시는 캐시한 데이터 집합을 최신 데이터로 유지하려고 쿼리 캐시를 실행하는 시간과 쿼리 캐시가 사용하는 테이블들이 가장 최근에 변경된 시간을 비교한다. 쿼리 캐시를 적용하고 난 후에 쿼리 캐시가 사용하는 테이블에 조금이라도 변경이 일어나면 데이터베이스에서 데이터를 읽어와서 쿼리 결과를 다시 캐시한다.

이제부터 엔티티를 변경하면 `org.hibernate.cache.spi.UpdateTimestampsCache` 영역에 해당 엔티티가 매핑한 테이블 이름으로 타임스탬프를 갱신한다.

예를 들어 다음과 같은 쿼리를 캐시한다고 가정해보자.

```
public List<ParentMember> findParentMembers() {
    return em.createQuery("select p from ParentMember p join p.childMembers c",
            .setHint("org.hibernate.cacheable", true)
            .getResultList();
}
```

예제는 쿼리에서 `ParentMember`와 `ChildMember` 엔티티를 사용한다.

쿼리를 실행하면 우선 `StandardQueryCache` 캐시 영역에서 타임스탬프를 조회한다. 그리고 쿼리가 사용하는 엔티티의 테이블인 `PARENTMEMBER`, `CHILDMEMBER`를 `UpdateTimestampsCache` 캐시 영역에서 조회해서 테이블들의 타임스탬프를 확인한다. 이때 만약 `StandardQueryCache` 캐시 영역의 타임스탬프가 더 오래되었으면 캐시가 유효하지 않은 것으로 보고 데이터베이스에서 데이터를 조회해서 다시 캐시한다.

쿼리 캐시를 잘 활용하면 극적인 성능 향상이 있지만 빈번하게 변경이 있는 테이블에 사용하면 오히려 성능이 더 저하된다. 따라서 수정이 거의 일어나지 않는 테이블에 사용해야 효과를 볼 수 있다.

> **주의:** `org.hibernate.cache.spi.UpdateTimestampsCache` 쿼리 캐시 영역은 만료되지 않도록 설정해야 한다. 해당 영역이 만료되면 모든 쿼리 캐시가 무효화 된다. EHCACHE의 `eternal="true"` 옵션을 사용하면 캐시에서 삭제되지 않는다.
>
> ```
> <cache
>         name="org.hibernate.cache.spi.UpdateTimestampsCache"
>         maxElementsInMemory="10000"
>         eternal="true" />
> ```

- 쿼리 캐시와 컬렉션 캐시의 주의점

엔티티 캐시를 사용해서 엔티티를 캐시하면 엔티티 정보를 모두 캐시하지만 쿼리 캐시와 컬렉션 캐시는 결과 집합의 **식별자 값만 캐시**한다. 따라서 쿼리 캐시와 컬렉션 캐시를 조회(캐시 히트)하면 그 안에는 사실 식별자 값만 들어 있다. 그리고 이 식별자 값을 하나씩 엔티티 캐시에서 조회해서 실제 엔티티를 찾는다.

문제는 쿼리 캐시나 컬렉션 캐시만 사용하고 대상 엔티티에 앤티티 캐시를 적용하지 않으면 성능상 심각한 문제가 발생할 수 있다. 예를 들어 보자.

1. `select m from Member m` 쿼리를 실행했는데 쿼리 캐시가 적용되어 있다. 결과 집합은 100건이다.
2. 결과 집합에는 식별자만 있으므로 한 건씩 엔티티 캐시 영역에서 조회한다.
3. `Member` 엔티티는 엔티티 캐시를 사용하지 않으므로 한 건씩 데이터베이스에서 조회한다.
4. 결국 100건의 SQL이 실행된다.

쿼리 캐시나 컬렉션 캐시만 사용하고 엔티티 캐시를 사용하지 않으면 최악의 상황에 결과 집합 수만큼 SQL이 실행된다. **따라서 쿼리 캐시나 컬렉션 캐시를 사용하면 결과 대상 엔티티에는 꼭 엔티티 캐시를 적용해야 한다.**

---

1. http://ehcache.org/ ↩
2. http://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html_single/#d5e9236 ↩
3. http://ehcache.org/documentation ↩