C#中的多级缓存架构设计与实现深度解析

合集 - DotNet(3)

收起

C#中的多级缓存架构设计与实现深度解析

引言

在现代分布式应用架构中，缓存已成为提升系统性能和用户体验的关键技术组件。随着业务规模的不断扩大和并发量的持续增长，单一级别的缓存往求。多级缓存架构通过在不同层次构建缓存体系，能够显著提升数据访问效率，降低数据库负载，并提供更好的系统可扩展性。

本文将深入探讨C#环境下多级缓存的架构设计与实现，重点分析内存缓存（Memory Cache）与Redis分布式缓存的协同工作机制，并详细阐述如何通（Pub/Sub）模式实现不同节点间的缓存状态同步。

1. 多级缓存理论基础

1.1 缓存层次结构理论

缓存的本质是利用时间局部性（Temporal Locality）和空间局部性（Spatial Locality）原理，将频繁访问的数据存储在更快的存储介质中。在计算机系从内存到磁盘，都遵循着这种层次化的存储架构。

1.1.1 缓存访问模式

```
CPU Cache (L1/L2/L3) → Memory → Disk Storage
   ↑                     ↑         ↑
快速访问               中等速度    慢速访问
小容量                 中等容量    大容量
昂贵                   适中        便宜
```

在应用层面，多级缓存同样遵循类似的原理：

1. **L1缓存（进程内存缓存）**：访问速度最快，容量相对较小，仅在当前进程内有效
2. **L2缓存（分布式缓存）**：访问速度中等，容量较大，在多个节点间共享
3. **L3缓存（数据库查询缓存）**：访问速度最慢，但提供持久化存储

1.2 缓存一致性理论

1.2.1 CAP定理在缓存系统中的应用

根据CAP定理（Consistency, Availability, Partition tolerance），在分布式缓存系统中，我们无法同时保证：

- **一致性（Consistency）**：所有节点在同一时间具有相同的数据
- **可用性（Availability）**：系统持续可用，即使某些节点出现故障
- **分区容错性（Partition Tolerance）**：系统能够容忍网络分区故障

在实际应用中，我们通常采用**最终一致性（Eventually Consistency）**模型，通过合理的同步策略和过期机制来平衡性能与一致性。

1.2.2 缓存穿透、击穿、雪崩问题

缓存穿透（Cache Penetration）：

- 现象：查询不存在的数据，绕过缓存直接访问数据库
- 解决方案：布隆过滤器、空值缓存

缓存击穿（Cache Breakdown）：

- 现象：热点数据过期时，大量并发请求同时访问数据库
- 解决方案：分布式锁、热点数据永不过期

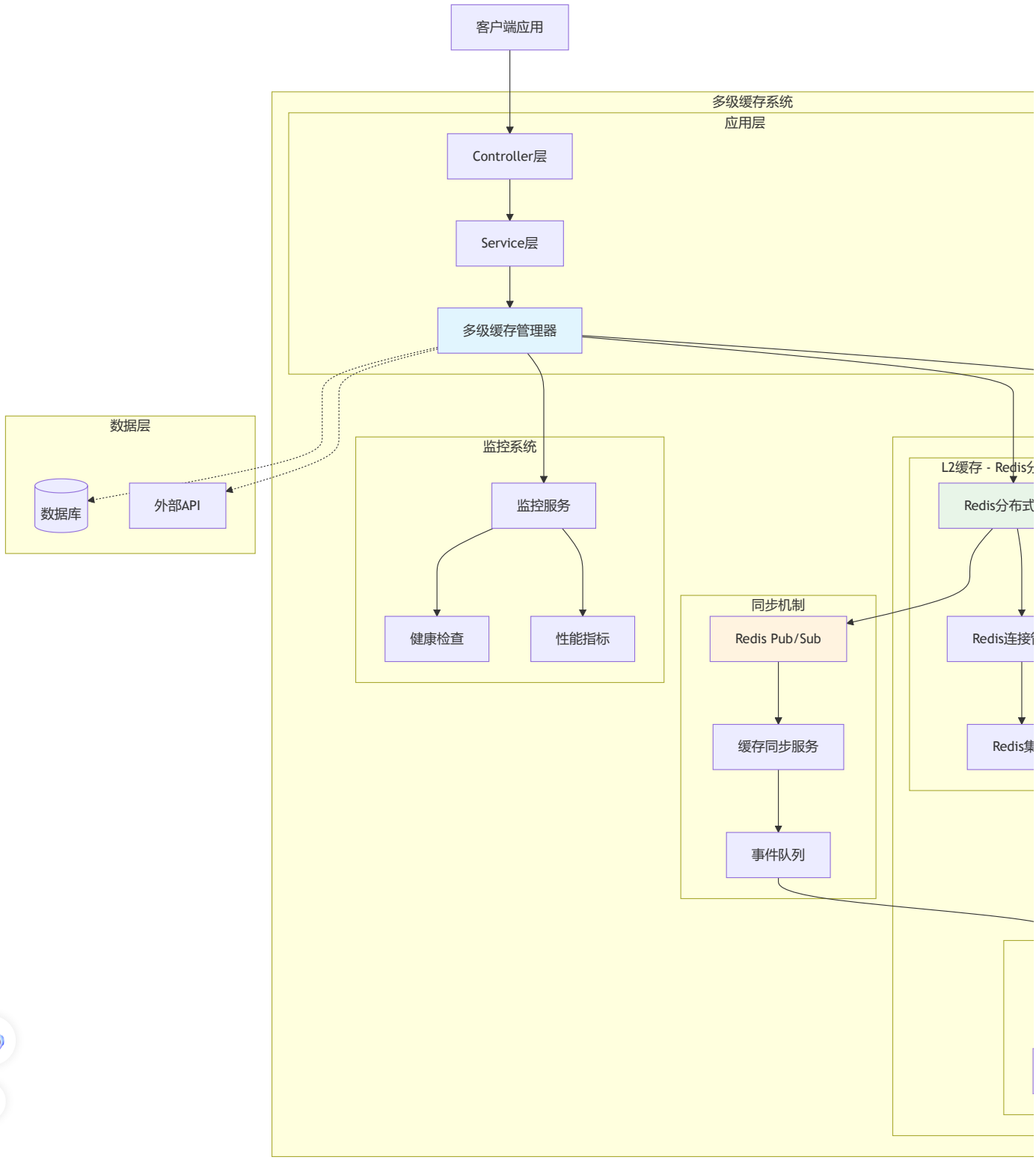
缓存雪崩（Cache Avalanche）：

- 现象：大量缓存同时失效，导致数据库压力骤增
- 解决方案：过期时间随机化、多级缓存、熔断机制

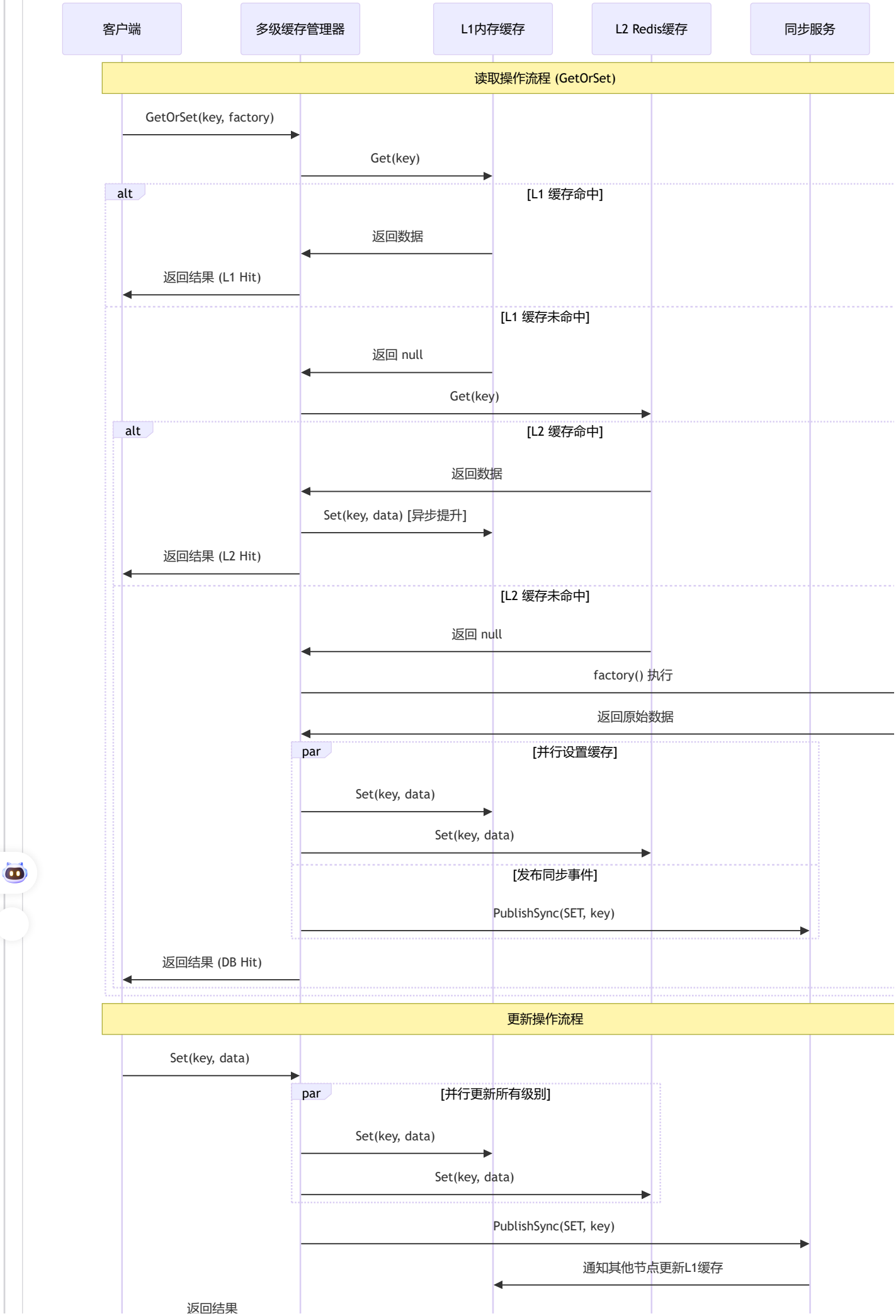
2. 架构设计与技术选型

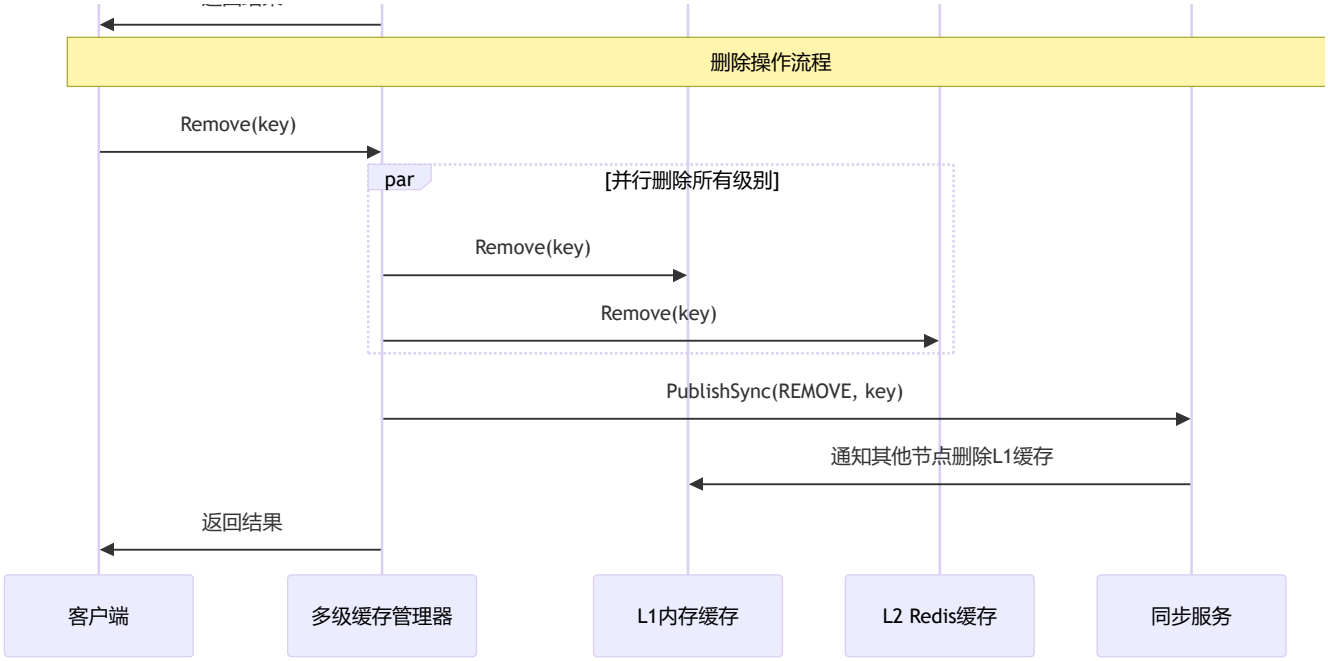
2.0 系统架构流程图

2.0.1 多级缓存整体架构

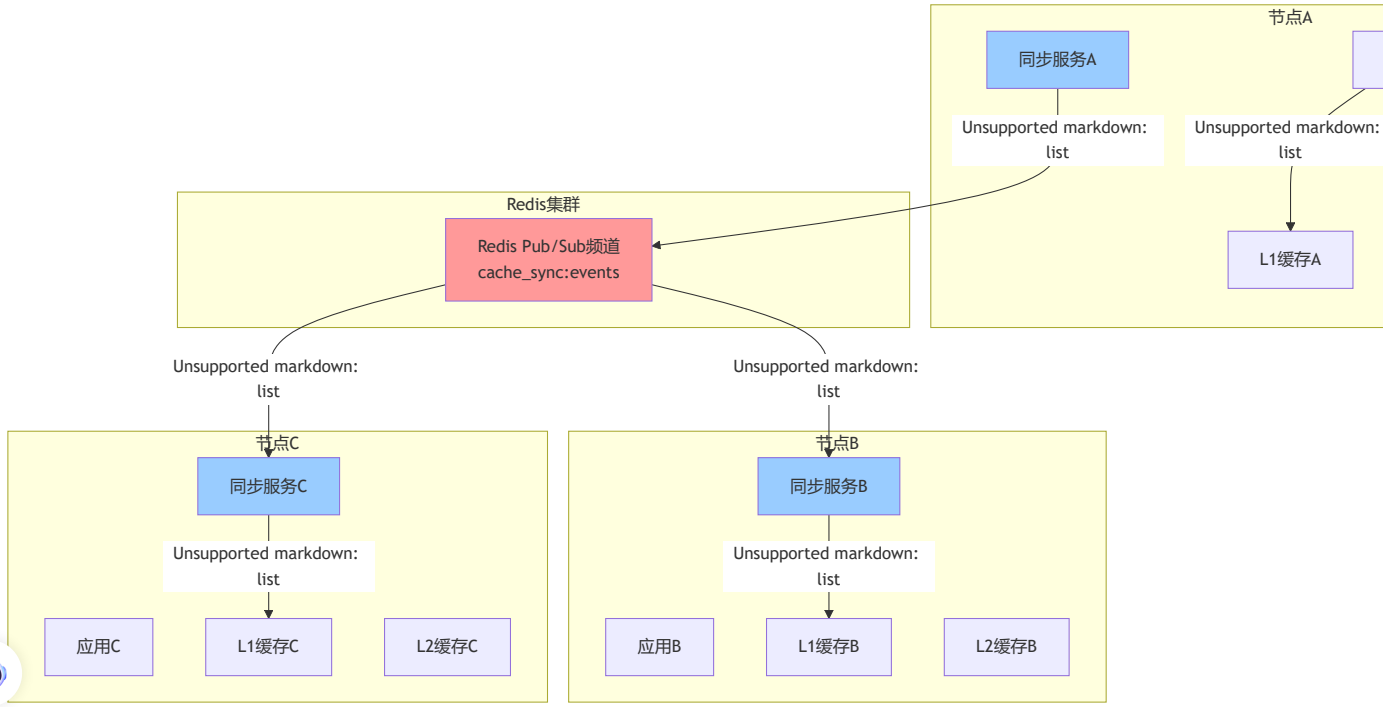

客户端应用
多级缓存系统
应用层
Controller层
Service层
多级缓存管理器
数据层
数据库
外部API
监控系统
监控服务
健康检查
性能指标
同步机制
Redis Pub/Sub
缓存同步服务
事件队列
Redis分布式
Redis连接
Redis集

2.0.2 缓存操作流程图

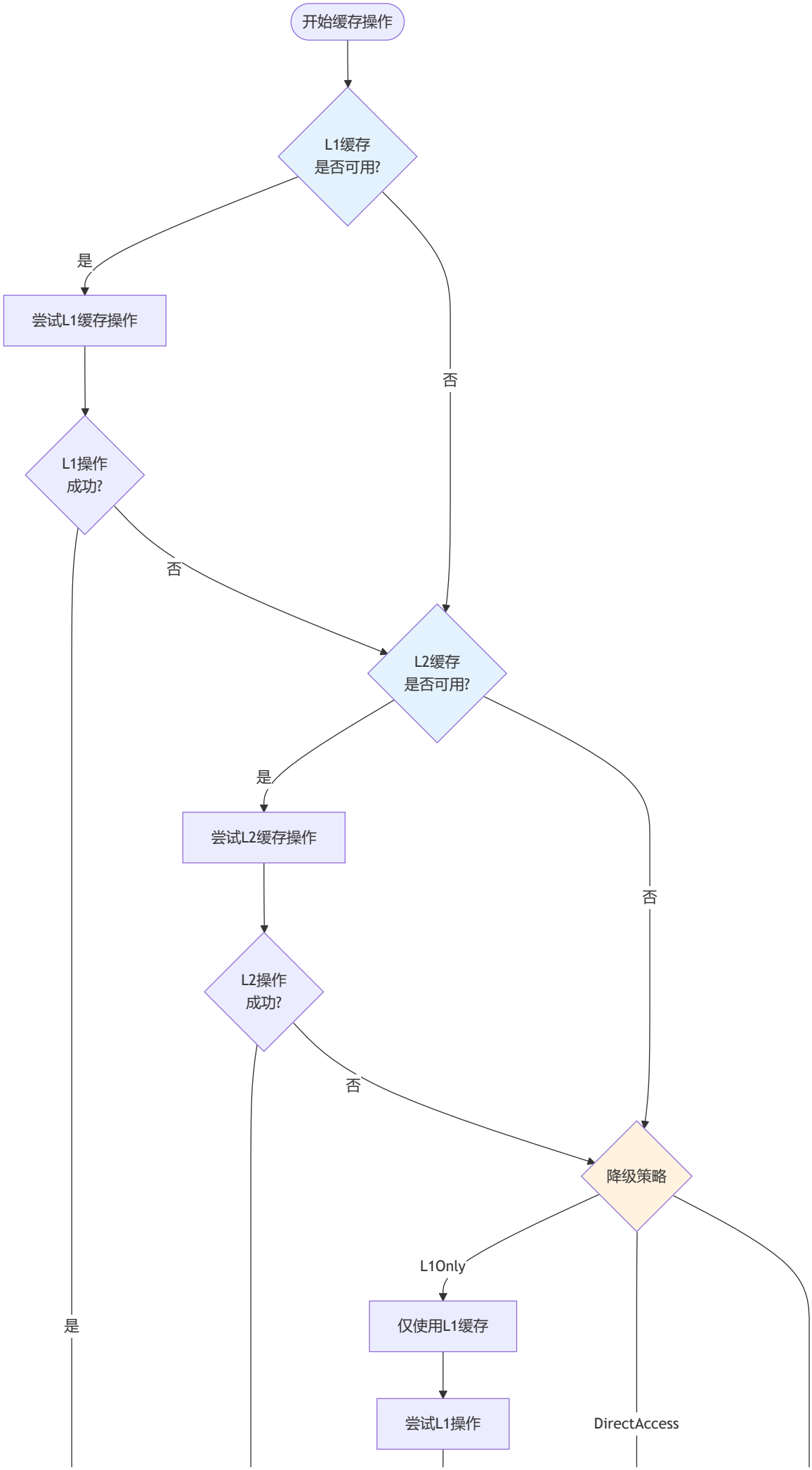

客户端
多级缓存管理器
L1内存缓存
L2 Redis缓存
同步服务
读取操作流程 (GetOrSet)
GetOrSet(key, factory)
Get(key)
alt
[L1 缓存命中]
返回数据
返回结果 (L1 Hit)
[L1 缓存未命中]
返回 null
Get(key)
alt
[L2 缓存命中]
返回数据
Set(key, data) [异步提升]
返回结果 (L2 Hit)
[L2 缓存未命中]
返回 null
factory() 执行
返回原始数据
par
[并行设置缓存]
Set(key, data)
Set(key, data)
[发布同步事件]
PublishSync(SET, key)
返回结果 (DB Hit)
更新操作流程
Set(key, data)
par
[并行更新所有级别]
Set(key, data)
Set(key, data)
PublishSync(SET, key)
通知其他节点更新L1缓存
返回结果

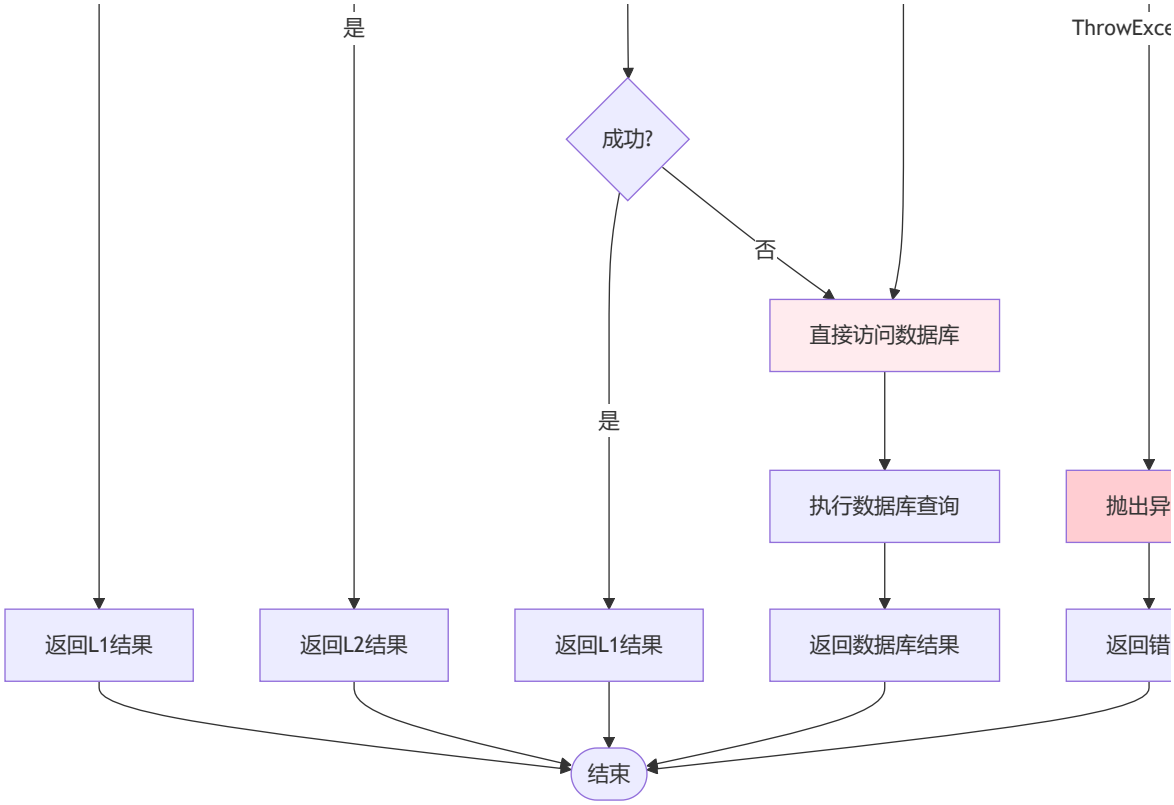


2.0.3 Redis发布-订阅同步机制



2.0.4 缓存降级策略流程


开始缓存操作
L1缓存
是否可用?
是
否
尝试L1缓存操作
L1操作
成功?
否
是
L2缓存
是否可用?
是
否
尝试L2缓存操作
L2操作
成功?
否
是
降级策略
L1Only
DirectAccess
仅使用L1缓存
尝试L1操作



2. 架构设计与技术选型

2.1 整体架构设计

多级缓存架构采用分层设计模式，每一层都有明确的职责和边界：

```
┌─────────────────────────────────────────────┐
│                应用层                       │
├─────────────────────────────────────────────┤
│            多级缓存管理器                   │
├──────────────────┬──────────────────────────┤
│ L1内存缓存       │ L2 Redis缓存             │
│ (MemoryCache)    │ (StackExchange.Redis)    │
├──────────────────┴──────────────────────────┤
│          Redis Pub/Sub 同步机制             │
├─────────────────────────────────────────────┤
│              数据持久层                     │
└─────────────────────────────────────────────┘
```

2.2 技术选型分析

2.2.1 内存缓存选型

Microsoft.Extensions.Caching.Memory：

- 优势：.NET官方支持，与DI容器无缝集成，支持过期策略和内存压力驱逐
- 适用场景：单体应用、微服务单实例缓存
- 特性：线程安全、支持泛型、内置压缩和序列化

System.Runtime.Caching.MemoryCache：

- 优势：.NET Framework传统方案，功能成熟
- 劣势：不支持.NET Core，API相对古老

2.2.2 分布式缓存选型

StackExchange.Redis：

- 优势：高性能、功能全面、支持集群、活跃的社区支持
- 特性：异步操作、连接复用、故障转移、Lua脚本支持

- 版本选择：推荐使用2.6+版本，支持.NET 6+的新特性

ServiceStack.Redis：

- 优势：易用性好，文档完善
- 劣势：商业许可限制，性能相对较低

2.3 架构模式选择

2.3.1 Cache-Aside Pattern（缓存旁路模式）

这是最常用的缓存模式，应用程序负责管理缓存的读取和更新：

```
读取流程：
1. 应用程序尝试从缓存读取数据
2. 如果缓存命中，直接返回数据
3. 如果缓存未命中，从数据库读取数据
4. 将数据写入缓存，然后返回给应用程序

更新流程：
1. 更新数据库
2. 删除或更新缓存中的对应数据
```

2.3.2 Write-Through Pattern（写透模式）

```
写入流程：
1. 应用程序写入缓存
2. 缓存服务同步写入数据库
3. 确认写入完成后返回成功
```

2.3.3 Write-Behind Pattern（写回模式）

```
写入流程：
1. 应用程序写入缓存
2. 立即返回成功
3. 缓存服务异步批量写入数据库
```

3. 内存缓存层实现详解

3.1 IMemoryCache 核心接口分析

Microsoft.Extensions.Caching.Memory.IMemoryCache接口提供了缓存操作的核心方法：

```
public interface IMemoryCache : IDisposable
{
    bool TryGetValue(object key, out object value);
    ICacheEntry CreateEntry(object key);
    void Remove(object key);
}
```

3.2 高级内存缓存封装实现

```
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using System;
using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;
using System.Text.RegularExpressions;
using System.Runtime.Serialization;

/// <summary>
/// 缓存异常基类
```

```
/// </summary>
public abstract class CacheException : Exception
{
    protected CacheException(string message) : base(message) { }
    protected CacheException(string message, Exception innerException) : base(message, innerException) { }
}

/// <summary>
/// 缓存连接异常
/// </summary>
public class CacheConnectionException : CacheException
{
    public CacheConnectionException(string message) : base(message) { }
    public CacheConnectionException(string message, Exception innerException) : base(message, innerException) { }
}

/// <summary>
/// 缓存序列化异常
/// </summary>
public class CacheSerializationException : CacheException
{
    public CacheSerializationException(string message) : base(message) { }
    public CacheSerializationException(string message, Exception innerException) : base(message, innerException) { }
}

/// <summary>
/// 缓存超时异常
/// </summary>
public class CacheTimeoutException : CacheException
{
    public CacheTimeoutException(string message) : base(message) { }
    public CacheTimeoutException(string message, Exception innerException) : base(message, innerException) { }
}

/// <summary>
/// 缓存验证异常
/// </summary>
public class CacheValidationException : CacheException
{
    public CacheValidationException(string message) : base(message) { }
    public CacheValidationException(string message, Exception innerException) : base(message, innerException) { }
}

/// <summary>
/// 线程安全的缓存统计追踪器
/// </summary>
public class CacheStatisticsTracker
{
    private long _totalOperations = 0;
    private long _l1Hits = 0;
    private long _l2Hits = 0;
    private long _totalMisses = 0;
    private readonly object _lock = new object();

    public void RecordOperation()
    {
        Interlocked.Increment(ref _totalOperations);
    }

    public void RecordHit(CacheLevel level)
    {
        switch (level)
        {
            case CacheLevel.L1:
                Interlocked.Increment(ref _l1Hits);
                break;
            case CacheLevel.L2:
                Interlocked.Increment(ref _l2Hits);
                break;
        }
    }
```

```
    public void RecordMiss()
    {
        Interlocked.Increment(ref _totalMisses);
    }

    public CacheStatisticsSnapshot GetSnapshot()
    {
        return new CacheStatisticsSnapshot
        {
            TotalOperations = Interlocked.Read(ref _totalOperations),
            L1Hits = Interlocked.Read(ref _l1Hits),
            L2Hits = Interlocked.Read(ref _l2Hits),
            TotalMisses = Interlocked.Read(ref _totalMisses)
        };
    }

    public void Reset()
    {
        lock (_lock)
        {
            Interlocked.Exchange(ref _totalOperations, 0);
            Interlocked.Exchange(ref _l1Hits, 0);
            Interlocked.Exchange(ref _l2Hits, 0);
            Interlocked.Exchange(ref _totalMisses, 0);
        }
    }
}

/// <summary>
/// 缓存统计快照
/// </summary>
public class CacheStatisticsSnapshot
{
    public long TotalOperations { get; init; }
    public long L1Hits { get; init; }
    public long L2Hits { get; init; }
    public long TotalMisses { get; init; }
    public long TotalHits => L1Hits + L2Hits;
    public double OverallHitRatio => TotalOperations == 0 ? 0 : (double)TotalHits / TotalOperations;
    public double L1HitRatio => TotalOperations == 0 ? 0 : (double)L1Hits / TotalOperations;
    public double L2HitRatio => TotalOperations == 0 ? 0 : (double)L2Hits / TotalOperations;
}

/// <summary>
/// 缓存数据验证器接口
/// </summary>
public interface ICacheDataValidator
{
    bool IsValid<T>(T value);
    void ValidateKey(string key);
    bool IsSafeForSerialization<T>(T value);
}

/// <summary>
/// 默认缓存数据验证器
/// </summary>
public class DefaultCacheDataValidator : ICacheDataValidator
{
    private readonly ILogger<DefaultCacheDataValidator> _logger;
    private readonly HashSet<Type> _forbiddenTypes;
    private readonly Regex _keyValidationRegex;

    public DefaultCacheDataValidator(ILogger<DefaultCacheDataValidator> logger)
    {
        _logger = logger;
        _forbiddenTypes = new HashSet<Type>
        {
            typeof(System.IO.FileStream),
            typeof(System.Net.Sockets.Socket),
            typeof(System.Threading.Thread),
            typeof(System.Threading.Tasks.Task)
        };
```

```
        // 限制key格式：只允许字母数字下划线冒号和点
        _keyValidationRegex = new Regex(@"^[a-zA-Z0-9_:.-]+$", RegexOptions.Compiled);
    }

    public bool IsValid<T>(T value)
    {
        if (value == null) return true;

        var valueType = value.GetType();

        // 检查禁止类型
        if (_forbiddenTypes.Contains(valueType))
        {
            _logger.LogWarning("Forbidden type in cache: {Type}", valueType.Name);
            return false;
        }

        // 检查循环引用（简化版）
        if (HasCircularReference(value))
        {
            _logger.LogWarning("Circular reference detected in cache value");
            return false;
        }

        return true;
    }

    public void ValidateKey(string key)
    {
        if (string.IsNullOrWhiteSpace(key))
            throw new CacheValidationException("Cache key cannot be null or empty");

        if (key.Length > 250)
            throw new CacheValidationException($"Cache key too long: {key.Length} characters");

        if (!_keyValidationRegex.IsMatch(key))
            throw new CacheValidationException($"Invalid characters in cache key: {key}");
    }

    public bool IsSafeForSerialization<T>(T value)
    {
        if (value == null) return true;

        var valueType = value.GetType();

        // 检查是否有序列化属性
        if (valueType.IsSerializable ||
            valueType.GetCustomAttributes(typeof(DataContractAttribute), false).Length > 0)
        {
            return true;
        }

        // 原始类型和字符串通常安全
        return valueType.IsPrimitive || valueType == typeof(string) || valueType == typeof(DateTime);
    }

    private bool HasCircularReference(object obj, HashSet<object> visited = null)
    {
        if (obj == null) return false;

        visited ??= new HashSet<object>();

        if (visited.Contains(obj))
            return true;

        visited.Add(obj);

        // 简化的循环检测，只检查一层
        var type = obj.GetType();
        if (type.IsPrimitive || type == typeof(string))
            return false;
```

```
        visited.Remove(obj);
        return false;
    }
}

/// <summary>
/// 安全缓存管理器装饰器
/// </summary>
public class SecureCacheManagerDecorator : IAdvancedMemoryCache
{
    private readonly IAdvancedMemoryCache _innerCache;
    private readonly ICacheDataValidator _validator;
    private readonly ILogger<SecureCacheManagerDecorator> _logger;

    public SecureCacheManagerDecorator(
        IAdvancedMemoryCache innerCache,
        ICacheDataValidator validator,
        ILogger<SecureCacheManagerDecorator> logger)
    {
        _innerCache = innerCache ?? throw new ArgumentNullException(nameof(innerCache));
        _validator = validator ?? throw new ArgumentNullException(nameof(validator));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public async Task<T> GetOrSetAsync<T>(string key, Func<Task<T>> factory, TimeSpan? expiry = null)
    {
        _validator.ValidateKey(key);
        return await _innerCache.GetOrSetAsync(key, async () =>
        {
            var value = await factory();
            if (!_validator.IsValid(value))
            {
                throw new CacheValidationException($"Invalid cache value for key: {key}");
            }
            return value;
        }, expiry);
    }

    public async Task<T> GetAsync<T>(string key)
    {
        _validator.ValidateKey(key);
        return await _innerCache.GetAsync<T>(key);
    }

    public async Task SetAsync<T>(string key, T value, TimeSpan? expiry = null)
    {
        _validator.ValidateKey(key);

        if (!_validator.IsValid(value))
        {
            throw new CacheValidationException($"Invalid cache value for key: {key}");
        }

        if (!_validator.IsSafeForSerialization(value))
        {
            _logger.LogWarning("Potentially unsafe serialization for key: {Key}, type: {Type}",
                key, value?.GetType().Name);
        }

        await _innerCache.SetAsync(key, value, expiry);
    }

    public async Task RemoveAsync(string key)
    {
        _validator.ValidateKey(key);
        await _innerCache.RemoveAsync(key);
    }

    public async Task RemoveByPatternAsync(string pattern)
    {
        if (string.IsNullOrWhiteSpace(pattern))
```

```
            throw new CacheValidationException("Pattern cannot be null or empty");

        await _innerCache.RemoveByPatternAsync(pattern);
    }

    public CacheStatistics GetStatistics() => _innerCache.GetStatistics();

    public void ClearStatistics() => _innerCache.ClearStatistics();
}

/// <summary>
/// 序列化器接口
/// </summary>
public interface ICacheSerializer
{
    byte[] Serialize<T>(T value);
    T Deserialize<T>(byte[] data);
    string SerializerName { get; }
    bool SupportsType(Type type);
}

/// <summary>
/// JSON序列化器（默认）
/// </summary>
public class JsonCacheSerializer : ICacheSerializer
{
    private readonly JsonSerializerOptions _options;

    public string SerializerName => "JSON";

    public JsonCacheSerializer()
    {
        _options = new JsonSerializerOptions
        {
            PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
            WriteIndented = false,
            DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull,
            PropertyNameCaseInsensitive = true
        };
    }

    public byte[] Serialize<T>(T value)
    {
        if (value == null) return null;
        if (typeof(T) == typeof(string)) return System.Text.Encoding.UTF8.GetBytes(value.ToString());

        var json = JsonSerializer.Serialize(value, _options);
        return System.Text.Encoding.UTF8.GetBytes(json);
    }

    public T Deserialize<T>(byte[] data)
    {
        if (data == null || data.Length == 0) return default(T);
        if (typeof(T) == typeof(string)) return (T)(object)System.Text.Encoding.UTF8.GetString(data);

        var json = System.Text.Encoding.UTF8.GetString(data);
        return JsonSerializer.Deserialize<T>(json, _options);
    }

    public bool SupportsType(Type type)
    {
        return true; // JSON支持所有类型
    }
}

/// <summary>
/// 二进制序列化器（用于简单类型）
/// </summary>
public class BinaryCacheSerializer : ICacheSerializer
{
    public string SerializerName => "Binary";
```

```
    private static readonly HashSet<Type> SupportedTypes = new()
    {
        typeof(int), typeof(long), typeof(double), typeof(float),
        typeof(bool), typeof(byte), typeof(short),
        typeof(DateTime), typeof(DateTimeOffset), typeof(TimeSpan),
        typeof(Guid), typeof(decimal)
    };

    public byte[] Serialize<T>(T value)
    {
        if (value == null) return null;

        var type = typeof(T);

        // 专门处理常见类型，提高性能
        return type switch
        {
            _ when type == typeof(int) => BitConverter.GetBytes((int)(object)value),
            _ when type == typeof(long) => BitConverter.GetBytes((long)(object)value),
            _ when type == typeof(double) => BitConverter.GetBytes((double)(object)value),
            _ when type == typeof(float) => BitConverter.GetBytes((float)(object)value),
            _ when type == typeof(bool) => BitConverter.GetBytes((bool)(object)value),
            _ when type == typeof(DateTime) => BitConverter.GetBytes(((DateTime)(object)value).ToBinary()),
            _ when type == typeof(Guid) => ((Guid)(object)value).ToByteArray(),
            _ when type == typeof(string) => System.Text.Encoding.UTF8.GetBytes(value.ToString()),
            _ => throw new NotSupportedException($"Type {type.Name} is not supported by BinaryCacheSerializer")
        };
    }

    public T Deserialize<T>(byte[] data)
    {
        if (data == null || data.Length == 0) return default(T);

        var type = typeof(T);

        object result = type switch
        {
            _ when type == typeof(int) => BitConverter.ToInt32(data, 0),
            _ when type == typeof(long) => BitConverter.ToInt64(data, 0),
            _ when type == typeof(double) => BitConverter.ToDouble(data, 0),
            _ when type == typeof(float) => BitConverter.ToSingle(data, 0),
            _ when type == typeof(bool) => BitConverter.ToBoolean(data, 0),
            _ when type == typeof(DateTime) => DateTime.FromBinary(BitConverter.ToInt64(data, 0)),
            _ when type == typeof(Guid) => new Guid(data),
            _ when type == typeof(string) => System.Text.Encoding.UTF8.GetString(data),
            _ => throw new NotSupportedException($"Type {type.Name} is not supported by BinaryCacheSerializer")
        };

        return (T)result;
    }

    public bool SupportsType(Type type)
    {
        return SupportedTypes.Contains(type) || type == typeof(string);
    }
}

/// <summary>
/// 智能序列化器管理器
/// </summary>
public class SmartCacheSerializer : ICacheSerializer
{
    private readonly ICacheSerializer[] _serializers;
    private readonly ILogger<SmartCacheSerializer> _logger;

    public string SerializerName => "Smart";

    public SmartCacheSerializer(ILogger<SmartCacheSerializer> logger)
    {
        _logger = logger;
        _serializers = new ICacheSerializer[]
        {
```

```
            new BinaryCacheSerializer(), // 优先使用二进制序列化
            new JsonCacheSerializer()    // 备选JSON序列化
        };
    }

    public byte[] Serialize<T>(T value)
    {
        if (value == null) return null;

        var type = typeof(T);

        foreach (var serializer in _serializers)
        {
            if (serializer.SupportsType(type))
            {
                try
                {
                    var data = serializer.Serialize(value);
                    // 在数据开头添加序列化器标识
                    var header = System.Text.Encoding.UTF8.GetBytes(serializer.SerializerName.PadRight(8));
                    var result = new byte[header.Length + data.Length];
                    Array.Copy(header, 0, result, 0, header.Length);
                    Array.Copy(data, 0, result, header.Length, data.Length);

                    return result;
                }
                catch (Exception ex)
                {
                    _logger.LogWarning(ex, "Serializer {SerializerName} failed for type {TypeName}",
                        serializer.SerializerName, type.Name);
                    continue;
                }
            }
        }

        throw new CacheSerializationException($"No suitable serializer found for type: {type.Name}");
    }

    public T Deserialize<T>(byte[] data)
    {
        if (data == null || data.Length < 8) return default(T);

        // 读取序列化器标识
        var headerBytes = new byte[8];
        Array.Copy(data, 0, headerBytes, 0, 8);
        var serializerName = System.Text.Encoding.UTF8.GetString(headerBytes).Trim();

        // 获取实际数据
        var actualData = new byte[data.Length - 8];
        Array.Copy(data, 8, actualData, 0, actualData.Length);

        // 找到对应的序列化器
        var serializer = _serializers.FirstOrDefault(s => s.SerializerName == serializerName);
        if (serializer == null)
        {
            _logger.LogWarning("Unknown serializer: {SerializerName}, falling back to JSON", serializerName);
            serializer = new JsonCacheSerializer();
        }

        try
        {
            return serializer.Deserialize<T>(actualData);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed to deserialize with {SerializerName}", serializerName);
            throw new CacheSerializationException($"Deserialization failed with {serializerName}", ex);
        }
    }

    public bool SupportsType(Type type)
    {
```

```
        return _serializers.Any(s => s.SupportsType(type));
    }
}

/// <summary>
/// 断路器状态
/// </summary>
public enum CircuitBreakerState
{
    Closed,     // 正常状态
    Open,       // 断路器打开，拒绝请求
    HalfOpen    // 半开状态，允许少量请求通过
}

/// <summary>
/// 缓存断路器配置
/// </summary>
public class CacheCircuitBreakerOptions
{
    public int FailureThreshold { get; set; } = 5;                    // 连续失败阈值
    public TimeSpan OpenTimeout { get; set; } = TimeSpan.FromMinutes(1); // 断路器打开时间
    public int SuccessThreshold { get; set; } = 2;                    // 半开状态成功阈值
    public TimeSpan SamplingDuration { get; set; } = TimeSpan.FromMinutes(2); // 采样时间窗口
}

/// <summary>
/// 缓存断路器
/// </summary>
public class CacheCircuitBreaker
{
    private readonly CacheCircuitBreakerOptions _options;
    private readonly ILogger<CacheCircuitBreaker> _logger;
    private readonly object _lock = new object();

    private CircuitBreakerState _state = CircuitBreakerState.Closed;
    private int _failureCount = 0;
    private int _successCount = 0;
    private DateTime _lastFailureTime = DateTime.MinValue;
    private DateTime _lastStateChangeTime = DateTime.UtcNow;

    public CacheCircuitBreaker(
        CacheCircuitBreakerOptions options,
        ILogger<CacheCircuitBreaker> logger)
    {
        _options = options ?? throw new ArgumentNullException(nameof(options));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public CircuitBreakerState State => _state;

    /// <summary>
    /// 执行带断路器保护的操作
    /// </summary>
    public async Task<T> ExecuteAsync<T>(Func<Task<T>> operation, string operationName = null)
    {
        if (!CanExecute())
        {
            throw new CacheException($"Circuit breaker is OPEN for operation: {operationName}");
        }

        try
        {
            var result = await operation();
            OnSuccess();
            return result;
        }
        catch (Exception ex)
        {
            OnFailure(ex, operationName);
            throw;
        }
    }
```

```
    /// <summary>
    /// 检查是否可以执行操作
    /// </summary>
    private bool CanExecute()
    {
        lock (_lock)
        {
            switch (_state)
            {
                case CircuitBreakerState.Closed:
                    return true;

                case CircuitBreakerState.Open:
                    // 检查是否可以转入半开状态
                    if (DateTime.UtcNow - _lastStateChangeTime >= _options.OpenTimeout)
                    {
                        _state = CircuitBreakerState.HalfOpen;
                        _successCount = 0;
                        _lastStateChangeTime = DateTime.UtcNow;
                        _logger.LogInformation("Circuit breaker entering HALF_OPEN state");
                        return true;
                    }
                    return false;

                case CircuitBreakerState.HalfOpen:
                    return true;

                default:
                    return false;
            }
        }
    }

    /// <summary>
    /// 操作成功回调
    /// </summary>
    private void OnSuccess()
    {
        lock (_lock)
        {
            if (_state == CircuitBreakerState.HalfOpen)
            {
                _successCount++;
                if (_successCount >= _options.SuccessThreshold)
                {
                    _state = CircuitBreakerState.Closed;
                    _failureCount = 0;
                    _successCount = 0;
                    _lastStateChangeTime = DateTime.UtcNow;
                    _logger.LogInformation("Circuit breaker entering CLOSED state");
                }
            }
            else if (_state == CircuitBreakerState.Closed)
            {
                // 在采样时间窗口内重置失败计数
                if (DateTime.UtcNow - _lastFailureTime > _options.SamplingDuration)
                {
                    _failureCount = 0;
                }
            }
        }
    }

    /// <summary>
    /// 操作失败回调
    /// </summary>
    private void OnFailure(Exception ex, string operationName)
    {
        lock (_lock)
        {
            _failureCount++;
```

```
            _lastFailureTime = DateTime.UtcNow;

            _logger.LogWarning(ex, "Circuit breaker recorded failure #{FailureCount} for operation: {Operation}",
                _failureCount, operationName);

            if (_state == CircuitBreakerState.Closed && _failureCount >= _options.FailureThreshold)
            {
                _state = CircuitBreakerState.Open;
                _lastStateChangeTime = DateTime.UtcNow;
                _logger.LogError("Circuit breaker entering OPEN state after {FailureCount} failures", _failureCount);
            }
            else if (_state == CircuitBreakerState.HalfOpen)
            {
                _state = CircuitBreakerState.Open;
                _lastStateChangeTime = DateTime.UtcNow;
                _logger.LogWarning("Circuit breaker returning to OPEN state from HALF_OPEN due to failure");
            }
        }
    }

    /// <summary>
    /// 获取当前状态信息
    /// </summary>
    public object GetState()
    {
        lock (_lock)
        {
            return new
            {
                State = _state.ToString(),
                FailureCount = _failureCount,
                SuccessCount = _successCount,
                LastFailureTime = _lastFailureTime,
                LastStateChangeTime = _lastStateChangeTime,
                CanExecute = CanExecute()
            };
        }
    }
}

/// <summary>
/// 带断路器的Redis缓存装饰器
/// </summary>
public class CircuitBreakerRedisCache : IRedisDistributedCache
{
    private readonly IRedisDistributedCache _innerCache;
    private readonly CacheCircuitBreaker _circuitBreaker;
    private readonly ILogger<CircuitBreakerRedisCache> _logger;

    public CircuitBreakerRedisCache(
        IRedisDistributedCache innerCache,
        CacheCircuitBreaker circuitBreaker,
        ILogger<CircuitBreakerRedisCache> logger)
    {
        _innerCache = innerCache ?? throw new ArgumentNullException(nameof(innerCache));
        _circuitBreaker = circuitBreaker ?? throw new ArgumentNullException(nameof(circuitBreaker));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public async Task<T> GetAsync<T>(string key)
    {
        try
        {
            return await _circuitBreaker.ExecuteAsync(() => _innerCache.GetAsync<T>(key), $"GET:{key}");
        }
        catch (CacheException) when (_circuitBreaker.State == CircuitBreakerState.Open)
        {
            _logger.LogWarning("Circuit breaker open, returning default for key: {Key}", key);
            return default(T);
        }
    }
```

```
    public async Task SetAsync<T>(string key, T value, TimeSpan? expiry = null)
    {
        try
        {
            await _circuitBreaker.ExecuteAsync(() => _innerCache.SetAsync(key, value, expiry), $"SET:{key}");
        }
        catch (CacheException) when (_circuitBreaker.State == CircuitBreakerState.Open)
        {
            _logger.LogWarning("Circuit breaker open, skipping cache set for key: {Key}", key);
            // 不继续抛出异常，允许应用继续运行
        }
    }

    // 继续实现其他接口方法...
    public Task<bool> ExistsAsync(string key) =>
        _circuitBreaker.ExecuteAsync(() => _innerCache.ExistsAsync(key), $"EXISTS:{key}");

    public Task<bool> RemoveAsync(string key) =>
        _circuitBreaker.ExecuteAsync(() => _innerCache.RemoveAsync(key), $"REMOVE:{key}");

    public Task<long> RemoveByPatternAsync(string pattern) =>
        _circuitBreaker.ExecuteAsync(() => _innerCache.RemoveByPatternAsync(pattern), $"REMOVE_PATTERN:{pattern}");

    public Task<T> GetOrSetAsync<T>(string key, Func<Task<T>> factory, TimeSpan? expiry = null) =>
        _circuitBreaker.ExecuteAsync(() => _innerCache.GetOrSetAsync(key, factory, expiry), $"GET_OR_SET:{key}");

    public Task<Dictionary<string, T>> GetMultipleAsync<T>(IEnumerable<string> keys) =>
        _circuitBreaker.ExecuteAsync(() => _innerCache.GetMultipleAsync<T>(keys), "GET_MULTIPLE");

    public Task SetMultipleAsync<T>(Dictionary<string, T> keyValuePairs, TimeSpan? expiry = null) =>
        _circuitBreaker.ExecuteAsync(() => _innerCache.SetMultipleAsync(keyValuePairs, expiry), "SET_MULTIPLE");

    public Task<long> IncrementAsync(string key, long value = 1, TimeSpan? expiry = null) =>
        _circuitBreaker.ExecuteAsync(() => _innerCache.IncrementAsync(key, value, expiry), $"INCREMENT:{key}");

    public Task<double> IncrementAsync(string key, double value, TimeSpan? expiry = null) =>
        _circuitBreaker.ExecuteAsync(() => _innerCache.IncrementAsync(key, value, expiry), $"INCREMENT_DOUBLE:{key}");

    public Task<bool> SetIfNotExistsAsync<T>(string key, T value, TimeSpan? expiry = null) =>
        _circuitBreaker.ExecuteAsync(() => _innerCache.SetIfNotExistsAsync(key, value, expiry), $"SET_IF_NOT_EXISTS:{key}");

    public Task<TimeSpan?> GetExpiryAsync(string key) =>
        _circuitBreaker.ExecuteAsync(() => _innerCache.GetExpiryAsync(key), $"GET_EXPIRY:{key}");

    public Task<bool> ExpireAsync(string key, TimeSpan expiry) =>
        _circuitBreaker.ExecuteAsync(() => _innerCache.ExpireAsync(key, expiry), $"EXPIRE:{key}");
}

/// <summary>
/// LRU缓存容器，用于防止内存泄漏
/// </summary>
public class LRUCache<TKey, TValue>
{
    private readonly int _maxSize;
    private readonly Dictionary<TKey, LinkedListNode<CacheItem<TKey, TValue>>> _cache;
    private readonly LinkedList<CacheItem<TKey, TValue>> _lruList;
    private readonly object _lock = new object();

    public LRUCache(int maxSize)
    {
        if (maxSize <= 0)
            throw new ArgumentException("Max size must be greater than 0", nameof(maxSize));

        _maxSize = maxSize;
        _cache = new Dictionary<TKey, LinkedListNode<CacheItem<TKey, TValue>>>(maxSize);
        _lruList = new LinkedList<CacheItem<TKey, TValue>>();
    }

    public int Count
    {
        get
        {
```

```
        lock (_lock)
        {
            return _cache.Count;
        }
    }
}

public bool TryGet(TKey key, out TValue value)
{
    lock (_lock)
    {
        if (_cache.TryGetValue(key, out var node))
        {
            // 移到链表头部（最近使用）
            _lruList.Remove(node);
            _lruList.AddFirst(node);

            value = node.Value.Value;
            return true;
        }

        value = default(TValue);
        return false;
    }
}

public void Add(TKey key, TValue value)
{
    lock (_lock)
    {
        if (_cache.TryGetValue(key, out var existingNode))
        {
            // 更新已存在的项
            existingNode.Value.Value = value;
            existingNode.Value.LastAccessed = DateTime.UtcNow;

            // 移到链表头部
            _lruList.Remove(existingNode);
            _lruList.AddFirst(existingNode);
        }
        else
        {
            // 检查容量限制
            if (_cache.Count >= _maxSize)
            {
                // 移除最久未使用的项
                var lastNode = _lruList.Last;
                if (lastNode != null)
                {
                    _cache.Remove(lastNode.Value.Key);
                    _lruList.RemoveLast();
                }
            }

            // 添加新项
            var newItem = new CacheItem<TKey, TValue>
            {
                Key = key,
                Value = value,
                LastAccessed = DateTime.UtcNow
            };

            var newNode = _lruList.AddFirst(newItem);
            _cache[key] = newNode;
        }
    }
}

public bool Remove(TKey key)
{
    lock (_lock)
    {
```

```
            if (_cache.TryGetValue(key, out var node))
            {
                _cache.Remove(key);
                _lruList.Remove(node);
                return true;
            }

            return false;
        }
    }

    public void Clear()
    {
        lock (_lock)
        {
            _cache.Clear();
            _lruList.Clear();
        }
    }

    public IEnumerable<TKey> Keys
    {
        get
        {
            lock (_lock)
            {
                return _cache.Keys.ToList();
            }
        }
    }

    /// <summary>
    /// 清理过期项
    /// </summary>
    public int CleanupExpired(TimeSpan maxAge)
    {
        var cutoffTime = DateTime.UtcNow - maxAge;
        var expiredKeys = new List<TKey>();

        lock (_lock)
        {
            foreach (var item in _lruList)
            {
                if (item.LastAccessed < cutoffTime)
                {
                    expiredKeys.Add(item.Key);
                }
            }

            foreach (var key in expiredKeys)
            {
                Remove(key);
            }
        }

        return expiredKeys.Count;
    }
}

/// <summary>
/// LRU缓存项
/// </summary>
class CacheItem<TKey, TValue>
{
    public TKey Key { get; set; }
    public TValue Value { get; set; }
    public DateTime LastAccessed { get; set; }
}

/// <summary>
/// 高级内存缓存管理器
/// 提供泛型支持、统计信息、性能监控等功能
```

```
/// </summary>
public interface IAdvancedMemoryCache
{
    Task<T> GetOrSetAsync<T>(string key, Func<Task<T>> factory, TimeSpan? expiry = null);
    Task<T> GetAsync<T>(string key);
    Task SetAsync<T>(string key, T value, TimeSpan? expiry = null);
    Task RemoveAsync(string key);
    Task RemoveByPatternAsync(string pattern);
    CacheStatistics GetStatistics();
    void ClearStatistics();
}

/// <summary>
/// 缓存统计信息
/// </summary>
public class CacheStatistics
{
    public long HitCount { get; set; }
    public long MissCount { get; set; }
    public long SetCount { get; set; }
    public long RemoveCount { get; set; }
    public double HitRatio => HitCount + MissCount == 0 ? 0 : (double)HitCount / (HitCount + MissCount);
    public DateTime StartTime { get; set; }
    public TimeSpan Duration => DateTime.UtcNow - StartTime;
}

/// <summary>
/// 缓存配置选项
/// </summary>
public class AdvancedMemoryCacheOptions
{
    public int SizeLimit { get; set; } = 1000;
    public TimeSpan DefaultExpiry { get; set; } = TimeSpan.FromMinutes(30);
    public bool EnableStatistics { get; set; } = true;
    public bool EnablePatternRemoval { get; set; } = true;
    public double CompactionPercentage { get; set; } = 0.1;
}

/// <summary>
/// 高级内存缓存实现
/// 基于IMemoryCache构建的功能增强版本
/// </summary>
public class AdvancedMemoryCache : IAdvancedMemoryCache, IDisposable
{
    private readonly IMemoryCache _cache;
    private readonly ILogger<AdvancedMemoryCache> _logger;
    private readonly AdvancedMemoryCacheOptions _options;
    private readonly CacheStatistics _statistics;
    private readonly ConcurrentDictionary<string, byte> _keyTracker;
    private readonly SemaphoreSlim _semaphore;
    private readonly Timer _cleanupTimer;

    public AdvancedMemoryCache(
        IMemoryCache cache,
        ILogger<AdvancedMemoryCache> logger,
        IOptions<AdvancedMemoryCacheOptions> options)
    {
        _cache = cache ?? throw new ArgumentNullException(nameof(cache));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
        _options = options?.Value ?? new AdvancedMemoryCacheOptions();

        _statistics = new CacheStatistics { StartTime = DateTime.UtcNow };
        _keyTracker = new ConcurrentDictionary<string, byte>();
        _semaphore = new SemaphoreSlim(1, 1);

        // 定期清理过期的key追踪记录
        _cleanupTimer = new Timer(CleanupKeyTracker, null,
            TimeSpan.FromMinutes(5), TimeSpan.FromMinutes(5));
    }

    /// <summary>
    /// 获取或设置缓存项
```

```
    /// 这是最常用的方法，实现了Cache-Aside模式
    /// </summary>
    /// <typeparam name="T">缓存项类型</typeparam>
    /// <param name="key">缓存键</param>
    /// <param name="factory">数据工厂方法</param>
    /// <param name="expiry">过期时间</param>
    /// <returns>缓存的值</returns>
    public async Task<T> GetOrSetAsync<T>(string key, Func<Task<T>> factory, TimeSpan? expiry = null)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        if (factory == null)
            throw new ArgumentNullException(nameof(factory));

        // 尝试从缓存获取
        var cachedValue = await GetAsync<T>(key);
        if (cachedValue != null)
        {
            _logger.LogDebug("Cache hit for key: {Key}", key);
            return cachedValue;
        }

        // 使用信号量防止并发执行相同的factory方法
        await _semaphore.WaitAsync();
        try
        {
            // 双重检查锁定模式
            cachedValue = await GetAsync<T>(key);
            if (cachedValue != null)
            {
                _logger.LogDebug("Cache hit on second check for key: {Key}", key);
                return cachedValue;
            }

            // 执行工厂方法获取数据
            _logger.LogDebug("Cache miss for key: {Key}, executing factory method", key);
            var value = await factory();

            // 将结果存入缓存
            await SetAsync(key, value, expiry);

            return value;
        }
        catch (CacheConnectionException ex)
        {
            _logger.LogWarning(ex, "Cache connection failed for key: {Key}, using fallback", key);
            // 缓存连接失败时，仍执行工厂方法但不缓存结果
            return await factory();
        }
        catch (CacheSerializationException ex)
        {
            _logger.LogError(ex, "Serialization failed for key: {Key}", key);
            throw;
        }
        catch (CacheTimeoutException ex)
        {
            _logger.LogWarning(ex, "Cache operation timeout for key: {Key}", key);
            return await factory();
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Unexpected error occurred while executing factory method for key: {Key}", key);
            throw new CacheException($"Cache operation failed for key: {key}", ex);
        }
        finally
        {
            _semaphore.Release();
        }
    }

    /// <summary>
```

```
    /// 异步获取缓存项
    /// </summary>
    /// <typeparam name="T">缓存项类型</typeparam>
    /// <param name="key">缓存键</param>
    /// <returns>缓存的值，如果不存在则返回默认值</returns>
    public Task<T> GetAsync<T>(string key)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        var found = _cache.TryGetValue(key, out var value);

        if (_options.EnableStatistics)
        {
            if (found)
                Interlocked.Increment(ref _statistics.HitCount);
            else
                Interlocked.Increment(ref _statistics.MissCount);
        }

        if (found && value is T typedValue)
        {
            return Task.FromResult(typedValue);
        }

        return Task.FromResult(default(T));
    }

    /// <summary>
    /// 异步设置缓存项
    /// </summary>
    /// <typeparam name="T">缓存项类型</typeparam>
    /// <param name="key">缓存键</param>
    /// <param name="value">缓存值</param>
    /// <param name="expiry">过期时间</param>
    public Task SetAsync<T>(string key, T value, TimeSpan? expiry = null)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        var cacheExpiry = expiry ?? _options.DefaultExpiry;

        using var entry = _cache.CreateEntry(key);
        entry.Value = value;
        entry.AbsoluteExpirationRelativeToNow = cacheExpiry;
        entry.Size = 1; // 简化的大小计算，实际应用中可根据对象大小设置

        // 设置过期回调
        entry.PostEvictionCallbacks.Add(new PostEvictionCallbackRegistration
        {
            EvictionCallback = OnCacheEntryEvicted,
            State = key
        });

        // 追踪缓存键
        if (_options.EnablePatternRemoval)
        {
            _keyTracker.TryAdd(key, 0);
        }

        if (_options.EnableStatistics)
        {
            Interlocked.Increment(ref _statistics.SetCount);
        }

        _logger.LogDebug("Set cache entry for key: {Key}, expiry: {Expiry}", key, cacheExpiry);

        return Task.CompletedTask;
    }

    /// <summary>
    /// 异步移除缓存项
```

```
    /// </summary>
    /// <param name="key">缓存键</param>
    public Task RemoveAsync(string key)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        _cache.Remove(key);
        _keyTracker.TryRemove(key, out _);

        if (_options.EnableStatistics)
        {
            Interlocked.Increment(ref _statistics.RemoveCount);
        }

        _logger.LogDebug("Removed cache entry for key: {Key}", key);

        return Task.CompletedTask;
    }

    /// <summary>
    /// 根据模式异步移除缓存项
    /// 支持通配符匹配，如 "user:*", "*:settings"
    /// </summary>
    /// <param name="pattern">匹配模式</param>
    public async Task RemoveByPatternAsync(string pattern)
    {
        if (string.IsNullOrEmpty(pattern))
            throw new ArgumentException("Pattern cannot be null or empty", nameof(pattern));

        if (!_options.EnablePatternRemoval)
        {
            _logger.LogWarning("Pattern removal is disabled");
            return;
        }

        var keysToRemove = new List<string>();
        var regexPattern = ConvertWildcardToRegex(pattern);
        var regex = new System.Text.RegularExpressions.Regex(regexPattern,
            System.Text.RegularExpressions.RegexOptions.IgnoreCase);

        foreach (var key in _keyTracker.Keys)
        {
            if (regex.IsMatch(key))
            {
                keysToRemove.Add(key);
            }
        }

        foreach (var key in keysToRemove)
        {
            await RemoveAsync(key);
        }

        _logger.LogInformation("Removed {Count} cache entries matching pattern: {Pattern}",
            keysToRemove.Count, pattern);
    }

    /// <summary>
    /// 获取缓存统计信息
    /// </summary>
    /// <returns>统计信息对象</returns>
    public CacheStatistics GetStatistics()
    {
        if (!_options.EnableStatistics)
        {
            return new CacheStatistics();
        }

        return new CacheStatistics
        {
            HitCount = _statistics.HitCount,
```

```
            MissCount = _statistics.MissCount,
            SetCount = _statistics.SetCount,
            RemoveCount = _statistics.RemoveCount,
            StartTime = _statistics.StartTime
        };
    }

    /// <summary>
    /// 清除统计信息
    /// </summary>
    public void ClearStatistics()
    {
        if (_options.EnableStatistics)
        {
            Interlocked.Exchange(ref _statistics.HitCount, 0);
            Interlocked.Exchange(ref _statistics.MissCount, 0);
            Interlocked.Exchange(ref _statistics.SetCount, 0);
            Interlocked.Exchange(ref _statistics.RemoveCount, 0);
            _statistics.StartTime = DateTime.UtcNow;
        }
    }

    /// <summary>
    /// 缓存项被驱逐时的回调方法
    /// </summary>
    /// <param name="key">缓存键</param>
    /// <param name="value">缓存值</param>
    /// <param name="reason">驱逐原因</param>
    /// <param name="state">状态对象</param>
    private void OnCacheEntryEvicted(object key, object value, EvictionReason reason, object state)
    {
        var cacheKey = state?.ToString();
        if (!string.IsNullOrEmpty(cacheKey))
        {
            _keyTracker.TryRemove(cacheKey, out _);
        }

        _logger.LogDebug("Cache entry evicted - Key: {Key}, Reason: {Reason}", key, reason);
    }

    /// <summary>
    /// 将通配符模式转换为正则表达式
    /// </summary>
    /// <param name="wildcardPattern">通配符模式</param>
    /// <returns>正则表达式字符串</returns>
    private static string ConvertWildcardToRegex(string wildcardPattern)
    {
        return "^" + System.Text.RegularExpressions.Regex.Escape(wildcardPattern)
            .Replace("\\*", ".*")
            .Replace("\\?", ".") + "$";
    }

    /// <summary>
    /// 定期清理key追踪器中的过期项
    /// </summary>
    /// <param name="state">定时器状态</param>
    private void CleanupKeyTracker(object state)
    {
        var keysToRemove = new List<string>();

        foreach (var key in _keyTracker.Keys)
        {
            if (!_cache.TryGetValue(key, out _))
            {
                keysToRemove.Add(key);
            }
        }

        foreach (var key in keysToRemove)
        {
            _keyTracker.TryRemove(key, out _);
        }
```

```
        if (keysToRemove.Count > 0)
        {
            _logger.LogDebug("Cleaned up {Count} expired keys from tracker", keysToRemove.Count);
        }
    }

    /// <summary>
    /// 释放资源
    /// </summary>
    public void Dispose()
    {
        _cleanupTimer?.Dispose();
        _semaphore?.Dispose();
        _cache?.Dispose();
    }
}
```

3.3 内存缓存配置和依赖注入

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

/// <summary>
/// 内存缓存服务扩展
/// </summary>
public static class MemoryCacheServiceExtensions
{
    /// <summary>
    /// 添加高级内存缓存服务
    /// </summary>
    /// <param name="services">服务集合</param>
    /// <param name="setupAction">配置委托</param>
    /// <returns>服务集合</returns>
    public static IServiceCollection AddAdvancedMemoryCache(
        this IServiceCollection services,
        Action<AdvancedMemoryCacheOptions> setupAction = null)
    {
        // 添加基础内存缓存
        services.AddMemoryCache(options =>
        {
            options.SizeLimit = 1000; // 设置缓存大小限制
            options.CompactionPercentage = 0.1; // 内存压力时的压缩百分比
            options.ExpirationScanFrequency = TimeSpan.FromMinutes(1); // 过期扫描频率
        });

        // 配置选项
        if (setupAction != null)
        {
            services.Configure(setupAction);
        }
        else
        {
            services.Configure<AdvancedMemoryCacheOptions>(options =>
            {
                // 默认配置
                options.SizeLimit = 1000;
                options.DefaultExpiry = TimeSpan.FromMinutes(30);
                options.EnableStatistics = true;
                options.EnablePatternRemoval = true;
                options.CompactionPercentage = 0.1;
            });
        }

        // 注册高级内存缓存服务（带安全装饰器）
        services.AddSingleton<AdvancedMemoryCache>();
        services.AddSingleton<IAdvancedMemoryCache>(provider =>
        {
            var innerCache = provider.GetRequiredService<AdvancedMemoryCache>();
            var validator = provider.GetRequiredService<ICacheDataValidator>();
```

```
            var logger = provider.GetRequiredService<ILogger<SecureCacheManagerDecorator>>();
            return new SecureCacheManagerDecorator(innerCache, validator, logger);
        });

        return services;
    }
}

/// <summary>
/// 示例：在Program.cs中的配置
/// </summary>
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // 添加增强组件
        builder.Services.AddSingleton<ICacheDataValidator, DefaultCacheDataValidator>();
        builder.Services.AddSingleton<ICacheSerializer, SmartCacheSerializer>();

        // 添加断路器配置
        builder.Services.Configure<CacheCircuitBreakerOptions>(options =>
        {
            options.FailureThreshold = 5;
            options.OpenTimeout = TimeSpan.FromMinutes(1);
            options.SuccessThreshold = 2;
        });
        builder.Services.AddSingleton<CacheCircuitBreaker>();

        // 添加高级内存缓存（带安全验证）
        builder.Services.AddAdvancedMemoryCache(options =>
        {
            options.SizeLimit = 2000;
            options.DefaultExpiry = TimeSpan.FromHours(1);
            options.EnableStatistics = true;
            options.EnablePatternRemoval = true;
            options.CompactionPercentage = 0.15;
        });

        var app = builder.Build();

        app.Run();
    }
}
```

. Redis分布式缓存层实现

4.1 Redis连接管理和配置

```
using StackExchange.Redis;
using Microsoft.Extensions.Options;
using Microsoft.Extensions.Logging;
using System.Text.Json;

/// <summary>
/// Redis缓存配置选项
/// </summary>
public class RedisCacheOptions
{
    public string ConnectionString { get; set; } = "localhost:6379";
    public int Database { get; set; } = 0;
    public string KeyPrefix { get; set; } = "app:";
    public TimeSpan DefaultExpiry { get; set; } = TimeSpan.FromHours(1);
    public int ConnectTimeout { get; set; } = 5000;
    public int SyncTimeout { get; set; } = 1000;
    public bool AllowAdmin { get; set; } = false;
    public string Password { get; set; }
    public bool Ssl { get; set; } = false;
```

```
    public int ConnectRetry { get; set; } = 3;
    public bool AbortOnConnectFail { get; set; } = false;
    public string ClientName { get; set; } = "MultiLevelCache";
}

/// <summary>
/// Redis连接管理器
/// 提供连接池管理和故障恢复功能
/// </summary>
public interface IRedisConnectionManager : IDisposable
{
    IDatabase GetDatabase();
    ISubscriber GetSubscriber();
    IServer GetServer();
    bool IsConnected { get; }
    Task<bool> TestConnectionAsync();
}

/// <summary>
/// Redis连接管理器实现
/// </summary>
public class RedisConnectionManager : IRedisConnectionManager
{
    private readonly RedisCacheOptions _options;
    private readonly ILogger<RedisConnectionManager> _logger;
    private readonly Lazy<ConnectionMultiplexer> _connectionMultiplexer;
    private bool _disposed = false;

    public RedisConnectionManager(
        IOptions<RedisCacheOptions> options,
        ILogger<RedisConnectionManager> logger)
    {
        _options = options?.Value ?? throw new ArgumentNullException(nameof(options));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));

        _connectionMultiplexer = new Lazy<ConnectionMultiplexer>(CreateConnection);
    }

    /// <summary>
    /// 创建Redis连接
    /// </summary>
    /// <returns>ConnectionMultiplexer实例</returns>
    private ConnectionMultiplexer CreateConnection()
    {
        var configurationOptions = new ConfigurationOptions
        {
            EndPoints = { _options.ConnectionString },
            ConnectTimeout = _options.ConnectTimeout,
            SyncTimeout = _options.SyncTimeout,
            AllowAdmin = _options.AllowAdmin,
            ConnectRetry = _options.ConnectRetry,
            AbortOnConnectFail = _options.AbortOnConnectFail,
            ClientName = _options.ClientName,
            Ssl = _options.Ssl
        };

        if (!string.IsNullOrEmpty(_options.Password))
        {
            configurationOptions.Password = _options.Password;
        }

        try
        {
            var connection = ConnectionMultiplexer.Connect(configurationOptions);

            // 注册连接事件
            connection.ConnectionFailed += OnConnectionFailed;
            connection.ConnectionRestored += OnConnectionRestored;
            connection.ErrorMessage += OnErrorMessage;
            connection.InternalError += OnInternalError;

            _logger.LogInformation("Redis connection established successfully");
```

```
            return connection;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed to establish Redis connection");
            throw;
        }
    }

    /// <summary>
    /// 获取数据库实例
    /// </summary>
    /// <returns>IDatabase实例</returns>
    public IDatabase GetDatabase()
    {
        return _connectionMultiplexer.Value.GetDatabase(_options.Database);
    }

    /// <summary>
    /// 获取订阅者实例
    /// </summary>
    /// <returns>ISubscriber实例</returns>
    public ISubscriber GetSubscriber()
    {
        return _connectionMultiplexer.Value.GetSubscriber();
    }

    /// <summary>
    /// 获取服务器实例
    /// </summary>
    /// <returns>IServer实例</returns>
    public IServer GetServer()
    {
        var endpoints = _connectionMultiplexer.Value.GetEndPoints();
        return _connectionMultiplexer.Value.GetServer(endpoints.First());
    }

    /// <summary>
    /// 检查连接状态
    /// </summary>
    public bool IsConnected => _connectionMultiplexer.IsValueCreated &&
                               _connectionMultiplexer.Value.IsConnected;

    /// <summary>
    /// 测试连接
    /// </summary>
    /// <returns>连接是否成功</returns>
    public async Task<bool> TestConnectionAsync()
    {
        try
        {
            var database = GetDatabase();
            await database.PingAsync();
            return true;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Redis connection test failed");
            return false;
        }
    }

    #region 事件处理

    private void OnConnectionFailed(object sender, ConnectionFailedEventArgs e)
    {
        _logger.LogError(e.Exception, "Redis connection failed: {EndPoint}", e.EndPoint);
    }

    private void OnConnectionRestored(object sender, ConnectionFailedEventArgs e)
    {
        _logger.LogInformation("Redis connection restored: {EndPoint}", e.EndPoint);
```

```
    }

    private void OnErrorMessage(object sender, RedisErrorEventArgs e)
    {
        _logger.LogError("Redis error: {Message}", e.Message);
    }

    private void OnInternalError(object sender, InternalErrorEventArgs e)
    {
        _logger.LogError(e.Exception, "Redis internal error");
    }

    #endregion

    /// <summary>
    /// 释放资源
    /// </summary>
    public void Dispose()
    {
        if (!_disposed)
        {
            if (_connectionMultiplexer.IsValueCreated)
            {
                _connectionMultiplexer.Value.Close();
                _connectionMultiplexer.Value.Dispose();
            }
            _disposed = true;
        }
    }
}
```

4.2 Redis分布式缓存服务实现

```
using StackExchange.Redis;
using System.Text.Json;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;

/// <summary>
/// Redis分布式缓存接口
/// </summary>
public interface IRedisDistributedCache
{
    Task<T> GetAsync<T>(string key);
    Task SetAsync<T>(string key, T value, TimeSpan? expiry = null);
    Task<bool> ExistsAsync(string key);
    Task<bool> RemoveAsync(string key);
    Task<long> RemoveByPatternAsync(string pattern);
    Task<T> GetOrSetAsync<T>(string key, Func<Task<T>> factory, TimeSpan? expiry = null);
    Task<Dictionary<string, T>> GetMultipleAsync<T>(IEnumerable<string> keys);
    Task SetMultipleAsync<T>(Dictionary<string, T> keyValuePairs, TimeSpan? expiry = null);
    Task<long> IncrementAsync(string key, long value = 1, TimeSpan? expiry = null);
    Task<double> IncrementAsync(string key, double value, TimeSpan? expiry = null);
    Task<bool> SetIfNotExistsAsync<T>(string key, T value, TimeSpan? expiry = null);
    Task<TimeSpan?> GetExpiryAsync(string key);
    Task<bool> ExpireAsync(string key, TimeSpan expiry);
}

/// <summary>
/// Redis分布式缓存实现
/// </summary>
public class RedisDistributedCache : IRedisDistributedCache
{
    private readonly IRedisConnectionManager _connectionManager;
    private readonly RedisCacheOptions _options;
    private readonly ILogger<RedisDistributedCache> _logger;
    private readonly ICacheSerializer _serializer;

    public RedisDistributedCache(
        IRedisConnectionManager connectionManager,
```

```
        IOptions<RedisCacheOptions> options,
        ILogger<RedisDistributedCache> logger)
    {
        _connectionManager = connectionManager ?? throw new ArgumentNullException(nameof(connectionManager));
        _options = options?.Value ?? throw new ArgumentNullException(nameof(options));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));

        // 配置JSON序列化选项
        // 使用智能序列化器替代直接的JSON序列化器
        _serializer = serviceProvider?.GetService<ICacheSerializer>() ?? new JsonCacheSerializer();
    }

    /// <summary>
    /// 异步获取缓存项
    /// </summary>
    /// <typeparam name="T">缓存项类型</typeparam>
    /// <param name="key">缓存键</param>
    /// <returns>缓存的值</returns>
    public async Task<T> GetAsync<T>(string key)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        try
        {
            var database = _connectionManager.GetDatabase();
            var fullKey = GetFullKey(key);
            var value = await database.StringGetAsync(fullKey);

            if (!value.HasValue)
            {
                _logger.LogDebug("Cache miss for key: {Key}", key);
                return default(T);
            }

            _logger.LogDebug("Cache hit for key: {Key}", key);
            return DeserializeValue<T>(value);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error getting value from Redis for key: {Key}", key);
            return default(T);
        }
    }

    /// <summary>
    /// 异步设置缓存项
    /// </summary>
    /// <typeparam name="T">缓存项类型</typeparam>
    /// <param name="key">缓存键</param>
    /// <param name="value">缓存值</param>
    /// <param name="expiry">过期时间</param>
    public async Task SetAsync<T>(string key, T value, TimeSpan? expiry = null)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        try
        {
            var database = _connectionManager.GetDatabase();
            var fullKey = GetFullKey(key);
            var serializedValue = SerializeValue(value);
            var expiration = expiry ?? _options.DefaultExpiry;

            await database.StringSetAsync(fullKey, serializedValue, expiration);
            _logger.LogDebug("Set cache value for key: {Key}, expiry: {Expiry}", key, expiration);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error setting value in Redis for key: {Key}", key);
            throw;
        }
```

```
    }

    /// <summary>
    /// 检查键是否存在
    /// </summary>
    /// <param name="key">缓存键</param>
    /// <returns>键是否存在</returns>
    public async Task<bool> ExistsAsync(string key)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        try
        {
            var database = _connectionManager.GetDatabase();
            var fullKey = GetFullKey(key);
            return await database.KeyExistsAsync(fullKey);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error checking existence in Redis for key: {Key}", key);
            return false;
        }
    }

    /// <summary>
    /// 异步移除缓存项
    /// </summary>
    /// <param name="key">缓存键</param>
    /// <returns>是否成功移除</returns>
    public async Task<bool> RemoveAsync(string key)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        try
        {
            var database = _connectionManager.GetDatabase();
            var fullKey = GetFullKey(key);
            var result = await database.KeyDeleteAsync(fullKey);

            _logger.LogDebug("Remove cache key: {Key}, success: {Success}", key, result);
            return result;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error removing key from Redis: {Key}", key);
            return false;
        }
    }

    /// <summary>
    /// 根据模式批量删除缓存项
    /// </summary>
    /// <param name="pattern">匹配模式</param>
    /// <returns>删除的项目数量</returns>
    public async Task<long> RemoveByPatternAsync(string pattern)
    {
        if (string.IsNullOrEmpty(pattern))
            throw new ArgumentException("Pattern cannot be null or empty", nameof(pattern));

        try
        {
            var server = _connectionManager.GetServer();
            var database = _connectionManager.GetDatabase();
            var fullPattern = GetFullKey(pattern);

            var keys = server.Keys(database.Database, fullPattern).ToArray();
            if (keys.Length == 0)
            {
                return 0;
            }
```

```
            var deletedCount = await database.KeyDeleteAsync(keys);
            _logger.LogInformation("Deleted {Count} keys matching pattern: {Pattern}", deletedCount, pattern);

            return deletedCount;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error removing keys by pattern from Redis: {Pattern}", pattern);
            return 0;
        }
    }

    /// <summary>
    /// 获取或设置缓存项（分布式锁实现）
    /// </summary>
    /// <typeparam name="T">缓存项类型</typeparam>
    /// <param name="key">缓存键</param>
    /// <param name="factory">数据工厂方法</param>
    /// <param name="expiry">过期时间</param>
    /// <returns>缓存的值</returns>
    public async Task<T> GetOrSetAsync<T>(string key, Func<Task<T>> factory, TimeSpan? expiry = null)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        if (factory == null)
            throw new ArgumentNullException(nameof(factory));

        // 尝试从缓存获取
        var cachedValue = await GetAsync<T>(key);
        if (cachedValue != null)
        {
            return cachedValue;
        }

        // 使用分布式锁防止缓存击穿
        var lockKey = $"{key}:lock";
        var lockValue = Guid.NewGuid().ToString();
        var database = _connectionManager.GetDatabase();

        try
        {
            // 尝试获取分布式锁
            var lockAcquired = await database.StringSetAsync(
                GetFullKey(lockKey),
                lockValue,
                TimeSpan.FromMinutes(1),
                When.NotExists);

            if (lockAcquired)
            {
                try
                {
                    // 再次检查缓存
                    cachedValue = await GetAsync<T>(key);
                    if (cachedValue != null)
                    {
                        return cachedValue;
                    }

                    // 执行工厂方法
                    _logger.LogDebug("Executing factory method for key: {Key}", key);
                    var value = await factory();

                    // 设置缓存
                    await SetAsync(key, value, expiry);

                    return value;
                }
                finally
                {
```

```
                    // 释放分布式锁（使用Lua脚本确保原子性）
                    const string releaseLockScript = @"
                        if redis.call('GET', KEYS[1]) == ARGV[1] then
                            return redis.call('DEL', KEYS[1])
                        else
                            return 0
                        end";

                    await database.ScriptEvaluateAsync(
                        releaseLockScript,
                        new RedisKey[] { GetFullKey(lockKey) },
                        new RedisValue[] { lockValue });
                }
            }
            else
            {
                // 等待锁释放并重试
                _logger.LogDebug("Waiting for lock to be released for key: {Key}", key);
                await Task.Delay(50); // 短暂等待

                // 重试获取缓存
                cachedValue = await GetAsync<T>(key);
                if (cachedValue != null)
                {
                    return cachedValue;
                }

                // 如果仍未获取到，执行降级策略
                _logger.LogWarning("Failed to acquire lock and cache miss for key: {Key}, executing factory method", key);
                return await factory();
            }
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error in GetOrSetAsync for key: {Key}", key);
            // 降级到直接执行工厂方法
            return await factory();
        }
    }

    /// <summary>
    /// 批量获取缓存项
    /// </summary>
    /// <typeparam name="T">缓存项类型</typeparam>
    /// <param name="keys">缓存键集合</param>
    /// <returns>键值对字典</returns>
    public async Task<Dictionary<string, T>> GetMultipleAsync<T>(IEnumerable<string> keys)
    {
        if (keys == null)
            throw new ArgumentNullException(nameof(keys));

        var keyList = keys.ToList();
        if (!keyList.Any())
        {
            return new Dictionary<string, T>();
        }

        try
        {
            var database = _connectionManager.GetDatabase();
            var fullKeys = keyList.Select(k => (RedisKey)GetFullKey(k)).ToArray();
            var values = await database.StringGetAsync(fullKeys);

            var result = new Dictionary<string, T>();
            for (int i = 0; i < keyList.Count; i++)
            {
                if (values[i].HasValue)
                {
                    result[keyList[i]] = DeserializeValue<T>(values[i]);
                }
            }
```

```
            _logger.LogDebug("Retrieved {Count} out of {Total} keys from Redis",
                result.Count, keyList.Count);

            return result;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error getting multiple values from Redis");
            return new Dictionary<string, T>();
        }
    }

    /// <summary>
    /// 批量设置缓存项
    /// </summary>
    /// <typeparam name="T">缓存项类型</typeparam>
    /// <param name="keyValuePairs">键值对字典</param>
    /// <param name="expiry">过期时间</param>
    public async Task SetMultipleAsync<T>(Dictionary<string, T> keyValuePairs, TimeSpan? expiry = null)
    {
        if (keyValuePairs == null || !keyValuePairs.Any())
            return;

        try
        {
            var database = _connectionManager.GetDatabase();
            var expiration = expiry ?? _options.DefaultExpiry;

            var tasks = keyValuePairs.Select(async kvp =>
            {
                var fullKey = GetFullKey(kvp.Key);
                var serializedValue = SerializeValue(kvp.Value);
                await database.StringSetAsync(fullKey, serializedValue, expiration);
            });

            await Task.WhenAll(tasks);

            _logger.LogDebug("Set {Count} cache values with expiry: {Expiry}",
                keyValuePairs.Count, expiration);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error setting multiple values in Redis");
            throw;
        }
    }

    /// <summary>
    /// 原子递增操作
    /// </summary>
    /// <param name="key">缓存键</param>
    /// <param name="value">递增值</param>
    /// <param name="expiry">过期时间</param>
    /// <returns>递增后的值</returns>
    public async Task<long> IncrementAsync(string key, long value = 1, TimeSpan? expiry = null)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        try
        {
            var database = _connectionManager.GetDatabase();
            var fullKey = GetFullKey(key);

            var result = await database.StringIncrementAsync(fullKey, value);

            if (expiry.HasValue)
            {
                await database.KeyExpireAsync(fullKey, expiry.Value);
            }

            return result;
```

```
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error incrementing value in Redis for key: {Key}", key);
            throw;
        }
    }

    /// <summary>
    /// 原子递增操作（浮点数）
    /// </summary>
    /// <param name="key">缓存键</param>
    /// <param name="value">递增值</param>
    /// <param name="expiry">过期时间</param>
    /// <returns>递增后的值</returns>
    public async Task<double> IncrementAsync(string key, double value, TimeSpan? expiry = null)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        try
        {
            var database = _connectionManager.GetDatabase();
            var fullKey = GetFullKey(key);

            var result = await database.StringIncrementAsync(fullKey, value);

            if (expiry.HasValue)
            {
                await database.KeyExpireAsync(fullKey, expiry.Value);
            }

            return result;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error incrementing double value in Redis for key: {Key}", key);
            throw;
        }
    }

    /// <summary>
    /// 仅在键不存在时设置值
    /// </summary>
    /// <typeparam name="T">缓存项类型</typeparam>
    /// <param name="key">缓存键</param>
    /// <param name="value">缓存值</param>
    /// <param name="expiry">过期时间</param>
    /// <returns>是否设置成功</returns>
    public async Task<bool> SetIfNotExistsAsync<T>(string key, T value, TimeSpan? expiry = null)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        try
        {
            var database = _connectionManager.GetDatabase();
            var fullKey = GetFullKey(key);
            var serializedValue = SerializeValue(value);
            var expiration = expiry ?? _options.DefaultExpiry;

            var result = await database.StringSetAsync(fullKey, serializedValue, expiration, When.NotExists);

            _logger.LogDebug("SetIfNotExists for key: {Key}, success: {Success}", key, result);
            return result;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error in SetIfNotExists for key: {Key}", key);
            return false;
        }
    }
```

```
    /// <summary>
    /// 获取键的过期时间
    /// </summary>
    /// <param name="key">缓存键</param>
    /// <returns>过期时间，如果键不存在或无过期时间则返回null</returns>
    public async Task<TimeSpan?> GetExpiryAsync(string key)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        try
        {
            var database = _connectionManager.GetDatabase();
            var fullKey = GetFullKey(key);
            return await database.KeyTimeToLiveAsync(fullKey);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error getting expiry for key: {Key}", key);
            return null;
        }
    }

    /// <summary>
    /// 设置键的过期时间
    /// </summary>
    /// <param name="key">缓存键</param>
    /// <param name="expiry">过期时间</param>
    /// <returns>是否设置成功</returns>
    public async Task<bool> ExpireAsync(string key, TimeSpan expiry)
    {
        if (string.IsNullOrEmpty(key))
            throw new ArgumentException("Key cannot be null or empty", nameof(key));

        try
        {
            var database = _connectionManager.GetDatabase();
            var fullKey = GetFullKey(key);
            return await database.KeyExpireAsync(fullKey, expiry);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error setting expiry for key: {Key}", key);
            return false;
        }
    }

    #region 辅助方法

    /// <summary>
    /// 获取完整的缓存键（带前缀）
    /// </summary>
    /// <param name="key">原始键</param>
    /// <returns>完整键</returns>
    private string GetFullKey(string key)
    {
        return $"{_options.KeyPrefix}{key}";
    }

    /// <summary>
    /// 序列化值
    /// </summary>
    /// <typeparam name="T">值类型</typeparam>
    /// <param name="value">要序列化的值</param>
    /// <returns>序列化后的字符串</returns>
    private string SerializeValue<T>(T value)
    {
        if (value == null) return null;

        try
        {
```

```
            var serializedBytes = _serializer.Serialize(value);
            return Convert.ToBase64String(serializedBytes);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error serializing value of type: {Type}", typeof(T).Name);
            throw new CacheSerializationException($"Failed to serialize value of type: {typeof(T).Name}", ex);
        }
    }

    /// <summary>
    /// 反序列化值
    /// </summary>
    /// <typeparam name="T">目标类型</typeparam>
    /// <param name="value">要反序列化的值</param>
    /// <returns>反序列化后的对象</returns>
    private T DeserializeValue<T>(string value)
    {
        if (string.IsNullOrEmpty(value)) return default(T);

        try
        {
            var serializedBytes = Convert.FromBase64String(value);
            return _serializer.Deserialize<T>(serializedBytes);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error deserializing value to type: {Type}", typeof(T).Name);
            throw new CacheSerializationException($"Failed to deserialize value to type: {typeof(T).Name}", ex);
        }
    }

    #endregion
}
```

5. Redis发布-订阅同步机制实现

5.1 缓存同步事件模型

```
using System.Text.Json;

/// <summary>
/// 缓存同步事件类型
/// </summary>
public enum CacheSyncEventType
{
    /// <summary>
    /// 缓存项被设置
    /// </summary>
    Set,

    /// <summary>
    /// 缓存项被删除
    /// </summary>
    Remove,

    /// <summary>
    /// 缓存项过期
    /// </summary>
    Expire,

    /// <summary>
    /// 批量删除（按模式）
    /// </summary>
    RemovePattern,

    /// <summary>
    /// 清空所有缓存
    /// </summary>
```

```
    Clear
}

/// <summary>
/// 缓存同步事件
/// </summary>
public class CacheSyncEvent
{
    /// <summary>
    /// 事件ID（用于幂等性控制）
    /// </summary>
    public string EventId { get; set; } = Guid.NewGuid().ToString();

    /// <summary>
    /// 事件类型
    /// </summary>
    public CacheSyncEventType EventType { get; set; }

    /// <summary>
    /// 缓存键
    /// </summary>
    public string Key { get; set; }

    /// <summary>
    /// 模式（用于批量删除）
    /// </summary>
    public string Pattern { get; set; }

    /// <summary>
    /// 事件发生时间
    /// </summary>
    public DateTime Timestamp { get; set; } = DateTime.UtcNow;

    /// <summary>
    /// 发起节点标识
    /// </summary>
    public string NodeId { get; set; }

    /// <summary>
    /// 附加数据
    /// </summary>
    public Dictionary<string, object> Metadata { get; set; } = new();

    /// <summary>
    /// 序列化为JSON
    /// </summary>
    /// <returns>JSON字符串</returns>
    public string ToJson()
    {
        return JsonSerializer.Serialize(this, new JsonSerializerOptions
        {
            PropertyNamingPolicy = JsonNamingPolicy.CamelCase
        });
    }

    /// <summary>
    /// 从JSON反序列化
    /// </summary>
    /// <param name="json">JSON字符串</param>
    /// <returns>缓存同步事件</returns>
    public static CacheSyncEvent FromJson(string json)
    {
        return JsonSerializer.Deserialize<CacheSyncEvent>(json, new JsonSerializerOptions
        {
            PropertyNamingPolicy = JsonNamingPolicy.CamelCase
        });
    }
}

/// <summary>
/// 缓存同步配置选项
/// </summary>
```

```
public class CacheSyncOptions
{
    /// <summary>
    /// Redis发布订阅频道前缀
    /// </summary>
    public string ChannelPrefix { get; set; } = "cache_sync";

    /// <summary>
    /// 当前节点ID
    /// </summary>
    public string NodeId { get; set; } = Environment.MachineName;

    /// <summary>
    /// 是否启用缓存同步
    /// </summary>
    public bool EnableSync { get; set; } = true;

    /// <summary>
    /// 事件去重窗口时间
    /// </summary>
    public TimeSpan DeduplicationWindow { get; set; } = TimeSpan.FromSeconds(30);

    /// <summary>
    /// 最大重试次数
    /// </summary>
    public int MaxRetryAttempts { get; set; } = 3;

    /// <summary>
    /// 重试延迟
    /// </summary>
    public TimeSpan RetryDelay { get; set; } = TimeSpan.FromSeconds(1);

    /// <summary>
    /// 批量处理的最大延迟
    /// </summary>
    public TimeSpan BatchMaxDelay { get; set; } = TimeSpan.FromMilliseconds(100);

    /// <summary>
    /// 批量处理的最大大小
    /// </summary>
    public int BatchMaxSize { get; set; } = 50;
}
```

5.2 Redis发布-订阅同步服务

```
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using StackExchange.Redis;
using System.Collections.Concurrent;
using System.Threading.Channels;

/// <summary>
/// 缓存同步服务接口
/// </summary>
public interface ICacheSyncService
{
    /// <summary>
    /// 发布缓存同步事件
    /// </summary>
    /// <param name="syncEvent">同步事件</param>
    Task PublishAsync(CacheSyncEvent syncEvent);

    /// <summary>
    /// 订阅缓存同步事件
    /// </summary>
    /// <param name="handler">事件处理器</param>
    Task SubscribeAsync(Func<CacheSyncEvent, Task> handler);

    /// <summary>
```

```
    /// 取消订阅
    /// </summary>
    Task UnsubscribeAsync();

    /// <summary>
    /// 检查服务状态
    /// </summary>
    bool IsHealthy { get; }
}

/// <summary>
/// Redis发布-订阅缓存同步服务
/// </summary>
public class RedisCacheSyncService : ICacheSyncService, IHostedService, IDisposable
{
    private readonly IRedisConnectionManager _connectionManager;
    private readonly CacheSyncOptions _options;
    private readonly ILogger<RedisCacheSyncService> _logger;

    // 事件处理器集合
    private readonly ConcurrentBag<Func<CacheSyncEvent, Task>> _handlers;

    // 事件去重缓存
    private readonly ConcurrentDictionary<string, DateTime> _processedEvents;

    // 批量处理通道
    private readonly Channel<CacheSyncEvent> _eventChannel;
    private readonly ChannelWriter<CacheSyncEvent> _eventWriter;
    private readonly ChannelReader<CacheSyncEvent> _eventReader;

    // 订阅和处理任务
    private Task _subscriptionTask;
    private Task _processingTask;
    private CancellationTokenSource _cancellationTokenSource;

    // 服务状态
    private volatile bool _isHealthy = false;
    private bool _disposed = false;

    public RedisCacheSyncService(
        IRedisConnectionManager connectionManager,
        IOptions<CacheSyncOptions> options,
        ILogger<RedisCacheSyncService> logger)
    {
        _connectionManager = connectionManager ?? throw new ArgumentNullException(nameof(connectionManager));
        _options = options?.Value ?? throw new ArgumentNullException(nameof(options));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));

        _handlers = new ConcurrentBag<Func<CacheSyncEvent, Task>>();
        _processedEvents = new ConcurrentDictionary<string, DateTime>();

        // 创建有界通道用于批量处理
        var channelOptions = new BoundedChannelOptions(_options.BatchMaxSize * 2)
        {
            FullMode = BoundedChannelFullMode.Wait,
            SingleReader = true,
            SingleWriter = false
        };

        _eventChannel = Channel.CreateBounded<CacheSyncEvent>(channelOptions);
        _eventWriter = _eventChannel.Writer;
        _eventReader = _eventChannel.Reader;
    }

    /// <summary>
    /// 服务健康状态
    /// </summary>
    public bool IsHealthy => _isHealthy;

    /// <summary>
    /// 发布缓存同步事件
    /// </summary>
```

```
    /// <param name="syncEvent">同步事件</param>
    public async Task PublishAsync(CacheSyncEvent syncEvent)
    {
        if (!_options.EnableSync || syncEvent == null)
            return;

        try
        {
            // 设置节点ID
            syncEvent.NodeId = _options.NodeId;

            var subscriber = _connectionManager.GetSubscriber();
            var channel = GetChannelName();
            var message = syncEvent.ToJson();

            await subscriber.PublishAsync(channel, message);

            _logger.LogDebug("Published sync event: {EventType} for key: {Key}",
                syncEvent.EventType, syncEvent.Key);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed to publish sync event: {EventType} for key: {Key}",
                syncEvent.EventType, syncEvent.Key);
        }
    }

    /// <summary>
    /// 订阅缓存同步事件
    /// </summary>
    /// <param name="handler">事件处理器</param>
    public Task SubscribeAsync(Func<CacheSyncEvent, Task> handler)
    {
        if (handler == null)
            throw new ArgumentNullException(nameof(handler));

        _handlers.Add(handler);
        _logger.LogDebug("Added sync event handler");

        return Task.CompletedTask;
    }

    /// <summary>
    /// 取消订阅
    /// </summary>
    public async Task UnsubscribeAsync()
    {
        try
        {
            if (_subscriptionTask != null && !_subscriptionTask.IsCompleted)
            {
                _cancellationTokenSource?.Cancel();
                await _subscriptionTask;
            }

            _logger.LogDebug("Unsubscribed from sync events");
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error during unsubscribe");
        }
    }

    /// <summary>
    /// 启动服务
    /// </summary>
    /// <param name="cancellationToken">取消令牌</param>
    public Task StartAsync(CancellationToken cancellationToken)
    {
        if (!_options.EnableSync)
        {
            _logger.LogInformation("Cache sync is disabled");
```

```
            return Task.CompletedTask;
        }

        _cancellationTokenSource = CancellationTokenSource.CreateLinkedTokenSource(cancellationToken);

        // 启动Redis订阅任务
        _subscriptionTask = StartSubscriptionAsync(_cancellationTokenSource.Token);

        // 启动事件处理任务
        _processingTask = StartProcessingAsync(_cancellationTokenSource.Token);

        // 启动清理任务
        _ = Task.Run(() => StartCleanupAsync(_cancellationTokenSource.Token), cancellationToken);

        _logger.LogInformation("Cache sync service started with NodeId: {NodeId}", _options.NodeId);
        return Task.CompletedTask;
    }

    /// <summary>
    /// 停止服务
    /// </summary>
    /// <param name="cancellationToken">取消令牌</param>
    public async Task StopAsync(CancellationToken cancellationToken)
    {
        _logger.LogInformation("Stopping cache sync service");

        _cancellationTokenSource?.Cancel();

        // 完成事件通道写入
        _eventWriter.TryComplete();

        try
        {
            // 等待订阅任务完成
            if (_subscriptionTask != null)
            {
                await _subscriptionTask.WaitAsync(TimeSpan.FromSeconds(5), cancellationToken);
            }

            // 等待处理任务完成
            if (_processingTask != null)
            {
                await _processingTask.WaitAsync(TimeSpan.FromSeconds(5), cancellationToken);
            }
        }
        catch (TimeoutException)
        {
            _logger.LogWarning("Cache sync service stop timed out");
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error stopping cache sync service");
        }

        _isHealthy = false;
        _logger.LogInformation("Cache sync service stopped");
    }

    /// <summary>
    /// 启动Redis订阅
    /// </summary>
    /// <param name="cancellationToken">取消令牌</param>
    private async Task StartSubscriptionAsync(CancellationToken cancellationToken)
    {
        var retryCount = 0;

        while (!cancellationToken.IsCancellationRequested && retryCount < _options.MaxRetryAttempts)
        {
            try
            {
                var subscriber = _connectionManager.GetSubscriber();
                var channel = GetChannelName();
```

```
            await subscriber.SubscribeAsync(channel, OnMessageReceived);

            _isHealthy = true;
            _logger.LogInformation("Successfully subscribed to Redis channel: {Channel}", channel);

            // 保持订阅状态
            while (!cancellationToken.IsCancellationRequested)
            {
                await Task.Delay(1000, cancellationToken);
            }

            break;
        }
        catch (OperationCanceledException)
        {
            break;
        }
        catch (Exception ex)
        {
            retryCount++;
            _isHealthy = false;

            _logger.LogError(ex, "Redis subscription failed, retry {RetryCount}/{MaxRetries}",
                retryCount, _options.MaxRetryAttempts);

            if (retryCount < _options.MaxRetryAttempts)
            {
                await Task.Delay(_options.RetryDelay, cancellationToken);
            }
        }
    }
}

/// <summary>
/// 启动事件批量处理
/// </summary>
/// <param name="cancellationToken">取消令牌</param>
private async Task StartProcessingAsync(CancellationToken cancellationToken)
{
    var eventBatch = new List<CacheSyncEvent>();
    var batchTimer = Stopwatch.StartNew();

    try
    {
        while (!cancellationToken.IsCancellationRequested)
        {
            // 尝试读取事件（带超时）
            var hasEvent = await _eventReader.WaitToReadAsync(cancellationToken);
            if (!hasEvent)
                break;

            // 收集批量事件
            while (_eventReader.TryRead(out var syncEvent) &&
                   eventBatch.Count < _options.BatchMaxSize)
            {
                eventBatch.Add(syncEvent);
            }

            // 检查是否应该处理批次
            var shouldProcess = eventBatch.Count >= _options.BatchMaxSize ||
                                batchTimer.Elapsed >= _options.BatchMaxDelay ||
                                !_eventReader.TryPeek(out _); // 没有更多事件

            if (shouldProcess && eventBatch.Count > 0)
            {
                await ProcessEventBatchAsync(eventBatch, cancellationToken);

                eventBatch.Clear();
                batchTimer.Restart();
            }
            else if (eventBatch.Count > 0)
```

```
                {
                    // 短暂等待以收集更多事件
                    await Task.Delay(10, cancellationToken);
                }
            }
        }
        catch (OperationCanceledException)
        {
            // 正常取消
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error in event processing loop");
        }
        finally
        {
            // 处理剩余的事件
            if (eventBatch.Count > 0)
            {
                try
                {
                    await ProcessEventBatchAsync(eventBatch, CancellationToken.None);
                }
                catch (Exception ex)
                {
                    _logger.LogError(ex, "Error processing final event batch");
                }
            }
        }
    }

    /// <summary>
    /// 处理事件批次
    /// </summary>
    /// <param name="events">事件列表</param>
    /// <param name="cancellationToken">取消令牌</param>
    private async Task ProcessEventBatchAsync(List<CacheSyncEvent> events, CancellationToken cancellationToken)
    {
        if (!events.Any())
            return;

        try
        {
            var tasks = events
                .Where(e => !IsEventProcessed(e.EventId))
                .Select(async e =>
                {
                    try
                    {
                        // 标记事件为已处理
                        MarkEventAsProcessed(e.EventId);

                        // 并行调用所有处理器
                        var handlerTasks = _handlers.Select(handler => handler(e));
                        await Task.WhenAll(handlerTasks);

                        _logger.LogDebug("Processed sync event: {EventType} for key: {Key}",
                            e.EventType, e.Key);
                    }
                    catch (Exception ex)
                    {
                        _logger.LogError(ex, "Error processing sync event: {EventId}", e.EventId);
                    }
                });

            await Task.WhenAll(tasks);

            _logger.LogDebug("Processed batch of {Count} sync events", events.Count);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error processing event batch");
```

```
        }
    }

    /// <summary>
    /// 处理接收到的Redis消息
    /// </summary>
    /// <param name="channel">频道</param>
    /// <param name="message">消息</param>
    private async void OnMessageReceived(RedisChannel channel, RedisValue message)
    {
        try
        {
            if (!message.HasValue)
                return;

            var syncEvent = CacheSyncEvent.FromJson(message);

            // 忽略自己发送的事件
            if (syncEvent.NodeId == _options.NodeId)
            {
                _logger.LogTrace("Ignoring self-generated event: {EventId}", syncEvent.EventId);
                return;
            }

            // 将事件加入处理队列
            if (!await _eventWriter.WaitToWriteAsync())
            {
                _logger.LogWarning("Event channel is closed, dropping event: {EventId}", syncEvent.EventId);
                return;
            }

            if (!_eventWriter.TryWrite(syncEvent))
            {
                _logger.LogWarning("Failed to queue sync event: {EventId}", syncEvent.EventId);
            }
        }
        catch (JsonException ex)
        {
            _logger.LogError(ex, "Failed to deserialize sync event message: {Message}", message.ToString());
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error processing received message");
        }
    }

    /// <summary>
    /// 检查事件是否已处理（防止重复处理）
    /// </summary>
    /// <param name="eventId">事件ID</param>
    /// <returns>是否已处理</returns>
    private bool IsEventProcessed(string eventId)
    {
        return _processedEvents.ContainsKey(eventId);
    }

    /// <summary>
    /// 标记事件为已处理
    /// </summary>
    /// <param name="eventId">事件ID</param>
    private void MarkEventAsProcessed(string eventId)
    {
        _processedEvents.TryAdd(eventId, DateTime.UtcNow);
    }

    /// <summary>
    /// 启动定期清理已处理事件记录
    /// </summary>
    /// <param name="cancellationToken">取消令牌</param>
    private async Task StartCleanupAsync(CancellationToken cancellationToken)
    {
        while (!cancellationToken.IsCancellationRequested)
```

```
        {
            try
            {
                await Task.Delay(TimeSpan.FromMinutes(5), cancellationToken);

                var cutoffTime = DateTime.UtcNow - _options.DeduplicationWindow;
                var keysToRemove = _processedEvents
                    .Where(kvp => kvp.Value < cutoffTime)
                    .Select(kvp => kvp.Key)
                    .ToList();

                foreach (var key in keysToRemove)
                {
                    _processedEvents.TryRemove(key, out _);
                }

                if (keysToRemove.Count > 0)
                {
                    _logger.LogDebug("Cleaned up {Count} processed event records", keysToRemove.Count);
                }
            }
            catch (OperationCanceledException)
            {
                break;
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Error in cleanup task");
            }
        }
    }

    /// <summary>
    /// 获取Redis频道名称
    /// </summary>
    /// <returns>频道名称</returns>
    private string GetChannelName()
    {
        return $"{_options.ChannelPrefix}:events";
    }

    /// <summary>
    /// 释放资源
    /// </summary>
    public void Dispose()
    {
        if (!_disposed)
        {
            _cancellationTokenSource?.Cancel();
            _eventWriter?.TryComplete();

            try
            {
                Task.WhenAll(
                    _subscriptionTask ?? Task.CompletedTask,
                    _processingTask ?? Task.CompletedTask
                ).Wait(TimeSpan.FromSeconds(5));
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Error during disposal");
            }

            _cancellationTokenSource?.Dispose();
            _disposed = true;
        }
    }
}
```

5.3 缓存同步扩展方法

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

/// <summary>
/// 缓存同步服务扩展方法
/// </summary>
public static class CacheSyncServiceExtensions
{
    /// <summary>
    /// 添加Redis缓存同步服务
    /// </summary>
    /// <param name="services">服务集合</param>
    /// <param name="setupAction">配置委托</param>
    /// <returns>服务集合</returns>
    public static IServiceCollection AddRedisCacheSync(
        this IServiceCollection services,
        Action<CacheSyncOptions> setupAction = null)
    {
        // 配置选项
        if (setupAction != null)
        {
            services.Configure(setupAction);
        }
        else
        {
            services.Configure<CacheSyncOptions>(options =>
            {
                // 使用默认配置
            });
        }

        // 注册Redis缓存服务（带断路器装饰器）
        services.AddSingleton<RedisDistributedCache>();
        services.AddSingleton<IRedisDistributedCache>(provider =>
        {
            var innerCache = provider.GetRequiredService<RedisDistributedCache>();
            var circuitBreaker = provider.GetRequiredService<CacheCircuitBreaker>();
            var logger = provider.GetRequiredService<ILogger<CircuitBreakerRedisCache>>();
            return new CircuitBreakerRedisCache(innerCache, circuitBreaker, logger);
        });

        // 注册缓存同步服务
        services.AddSingleton<ICacheSyncService, RedisCacheSyncService>();
        services.AddHostedService<RedisCacheSyncService>(provider =>
            (RedisCacheSyncService)provider.GetRequiredService<ICacheSyncService>());

        return services;
    }
}
```

6. 完整的多级缓存管理器实现

现在我将完成多级缓存管理器的实现，这是整个系统的核心组件：

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using System.Diagnostics;

/// <summary>
/// 多级缓存配置选项
/// </summary>
public class MultiLevelCacheOptions
{
    /// <summary>
    /// L1缓存（内存缓存）配置
    /// </summary>
    public AdvancedMemoryCacheOptions L1Options { get; set; } = new();
```

```
    /// <summary>
    /// L2缓存（Redis缓存）配置
    /// </summary>
    public RedisCacheOptions L2Options { get; set; } = new();

    /// <summary>
    /// 缓存同步配置
    /// </summary>
    public CacheSyncOptions SyncOptions { get; set; } = new();

    /// <summary>
    /// 是否启用L1缓存
    /// </summary>
    public bool EnableL1Cache { get; set; } = true;

    /// <summary>
    /// 是否启用L2缓存
    /// </summary>
    public bool EnableL2Cache { get; set; } = true;

    /// <summary>
    /// 是否启用缓存同步
    /// </summary>
    public bool EnableCacheSync { get; set; } = true;

    /// <summary>
    /// L1缓存与L2缓存的一致性策略
    /// </summary>
    public CacheConsistencyStrategy ConsistencyStrategy { get; set; } = CacheConsistencyStrategy.EventualConsistency;

    /// <summary>
    /// L2缓存回写延迟（用于Write-Behind模式）
    /// </summary>
    public TimeSpan L2WriteDelay { get; set; } = TimeSpan.FromSeconds(1);

    /// <summary>
    /// 降级策略：L2缓存不可用时的行为
    /// </summary>
    public CacheDegradationStrategy DegradationStrategy { get; set; } = CacheDegradationStrategy.L1Only;

    /// <summary>
    /// 健康检查间隔
    /// </summary>
    public TimeSpan HealthCheckInterval { get; set; } = TimeSpan.FromSeconds(30);

    /// <summary>
    /// 是否启用性能指标收集
    /// </summary>
    public bool EnableMetrics { get; set; } = true;
}

/// <summary>
/// 缓存一致性策略
/// </summary>
public enum CacheConsistencyStrategy
{
    /// <summary>
    /// 强一致性：所有操作同步到所有层级
    /// </summary>
    StrongConsistency,

    /// <summary>
    /// 最终一致性：异步同步，容忍短期不一致
    /// </summary>
    EventualConsistency,

    /// <summary>
    /// 会话一致性：同一会话内保证一致性
    /// </summary>
    SessionConsistency
}
```

```
/// <summary>
/// 缓存降级策略
/// </summary>
public enum CacheDegradationStrategy
{
    /// <summary>
    /// 仅使用L1缓存
    /// </summary>
    L1Only,

    /// <summary>
    /// 直接访问数据源
    /// </summary>
    DirectAccess,

    /// <summary>
    /// 抛出异常
    /// </summary>
    ThrowException
}

/// <summary>
/// 缓存操作上下文
/// </summary>
public class CacheOperationContext
{
    public string Key { get; set; }
    public string SessionId { get; set; }
    public bool ForceRefresh { get; set; }
    public TimeSpan? CustomExpiry { get; set; }
    public CacheLevel TargetLevel { get; set; } = CacheLevel.All;
    public Dictionary<string, object> Metadata { get; set; } = new();
}

/// <summary>
/// 缓存级别
/// </summary>
[Flags]
public enum CacheLevel
{
    None = 0,
    L1 = 1,
    L2 = 2,
    All = L1 | L2
}

/// <summary>
/// 缓存操作结果
/// </summary>
public class CacheOperationResult<T>
{
    public T Value { get; set; }
    public bool Success { get; set; }
    public CacheLevel HitLevel { get; set; }
    public TimeSpan Duration { get; set; }
    public string Error { get; set; }
    public CacheStatistics Statistics { get; set; }
}

/// <summary>
/// 多级缓存管理器接口
/// </summary>
public interface IMultiLevelCacheManager
{
    Task<CacheOperationResult<T>> GetAsync<T>(string key, CacheOperationContext context = null);
    Task<CacheOperationResult<T>> GetOrSetAsync<T>(string key, Func<Task<T>> factory, TimeSpan? expiry = null, CacheOperationCo
    Task<CacheOperationResult<bool>> SetAsync<T>(string key, T value, TimeSpan? expiry = null, CacheOperationContext context =
    Task<CacheOperationResult<bool>> RemoveAsync(string key, CacheOperationContext context = null);
    Task<CacheOperationResult<long>> RemoveByPatternAsync(string pattern, CacheOperationContext context = null);
    Task<CacheOperationResult<bool>> ExistsAsync(string key, CacheLevel level = CacheLevel.All);
    Task<MultiLevelCacheStatistics> GetStatisticsAsync();
    Task<bool> IsHealthyAsync();
```

```
    Task ClearAsync(CacheLevel level = CacheLevel.All);
}

/// <summary>
/// 多级缓存统计信息
/// </summary>
public class MultiLevelCacheStatistics
{
    public CacheStatistics L1Statistics { get; set; } = new();
    public CacheStatistics L2Statistics { get; set; } = new();
    public long TotalOperations { get; set; }
    public double OverallHitRatio { get; set; }
    public Dictionary<string, object> PerformanceMetrics { get; set; } = new();
    public DateTime CollectionTime { get; set; } = DateTime.UtcNow;
}

/// <summary>
/// 多级缓存管理器实现
/// </summary>
public class MultiLevelCacheManager : IMultiLevelCacheManager, IDisposable
{
    private readonly IAdvancedMemoryCache _l1Cache;
    private readonly IRedisDistributedCache _l2Cache;
    private readonly ICacheSyncService _syncService;
    private readonly MultiLevelCacheOptions _options;
    private readonly ILogger<MultiLevelCacheManager> _logger;

    // 性能计数器 - 线程安全的统计记录
    private readonly CacheStatisticsTracker _statisticsTracker = new();
    private readonly object _statsLock = new object();

    // 健康状态监控
    private volatile bool _l2HealthStatus = true;
    private readonly Timer _healthCheckTimer;

    // 同步状态管理 - 使用LRU防止内存泄漏
    private readonly LRUCache<string, DateTime> _recentUpdates = new(10000);

    // 降级状态
    private volatile bool _isDegraded = false;
    private DateTime _degradationStartTime;

    // 资源释放标识
    private bool _disposed = false;

    public MultiLevelCacheManager(
        IAdvancedMemoryCache l1Cache,
        IRedisDistributedCache l2Cache,
        ICacheSyncService syncService,
        IOptions<MultiLevelCacheOptions> options,
        ILogger<MultiLevelCacheManager> logger)
    {
        _l1Cache = l1Cache ?? throw new ArgumentNullException(nameof(l1Cache));
        _l2Cache = l2Cache ?? throw new ArgumentNullException(nameof(l2Cache));
        _syncService = syncService ?? throw new ArgumentNullException(nameof(syncService));
        _options = options?.Value ?? throw new ArgumentNullException(nameof(options));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));

        // 订阅缓存同步事件
        if (_options.EnableCacheSync)
        {
            _ = _syncService.SubscribeAsync(OnCacheSyncEventReceived);
        }

        // 启动健康检查定时器
        _healthCheckTimer = new Timer(PerformHealthCheck, null,
            TimeSpan.Zero, _options.HealthCheckInterval);

        _logger.LogInformation("MultiLevel cache manager initialized with L1: {L1Enabled}, L2: {L2Enabled}, Sync: {SyncEnabled}
            _options.EnableL1Cache, _options.EnableL2Cache, _options.EnableCacheSync);
    }
```

```
/// <summary>
/// 异步获取缓存项
/// </summary>
public async Task<CacheOperationResult<T>> GetAsync<T>(string key, CacheOperationContext context = null)
{
    var stopwatch = Stopwatch.StartNew();
    context ??= new CacheOperationContext { Key = key };

    _statisticsTracker.RecordOperation();

    try
    {
        // L1缓存查找
        if (_options.EnableL1Cache && (context.TargetLevel & CacheLevel.L1) != 0)
        {
            var l1Result = await _l1Cache.GetAsync<T>(key);
            if (l1Result != null)
            {
                _statisticsTracker.RecordHit(CacheLevel.L1);
                _logger.LogDebug("L1 cache hit for key: {Key}", key);

                return new CacheOperationResult<T>
                {
                    Value = l1Result,
                    Success = true,
                    HitLevel = CacheLevel.L1,
                    Duration = stopwatch.Elapsed
                };
            }
        }

        // L2缓存查找
        if (_options.EnableL2Cache && (context.TargetLevel & CacheLevel.L2) != 0 && _l2HealthStatus)
        {
            var l2Result = await _l2Cache.GetAsync<T>(key);
            if (l2Result != null)
            {
                _statisticsTracker.RecordHit(CacheLevel.L2);
                _logger.LogDebug("L2 cache hit for key: {Key}", key);

                // 将L2结果提升到L1缓存
                if (_options.EnableL1Cache)
                {
                    _ = Task.Run(async () =>
                    {
                        try
                        {
                            await _l1Cache.SetAsync(key, l2Result, context.CustomExpiry);
                            _logger.LogTrace("Promoted key to L1 cache: {Key}", key);
                        }
                        catch (Exception ex)
                        {
                            _logger.LogWarning(ex, "Failed to promote key to L1 cache: {Key}", key);
                        }
                    });
                }

                return new CacheOperationResult<T>
                {
                    Value = l2Result,
                    Success = true,
                    HitLevel = CacheLevel.L2,
                    Duration = stopwatch.Elapsed
                };
            }
        }

        // 缓存完全未命中
        _statisticsTracker.RecordMiss();
        _logger.LogDebug("Cache miss for key: {Key}", key);

        return new CacheOperationResult<T>
```

```
            {
                Success = false,
                HitLevel = CacheLevel.None,
                Duration = stopwatch.Elapsed
            };
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error during cache get operation for key: {Key}", key);

            return new CacheOperationResult<T>
            {
                Success = false,
                Error = ex.Message,
                Duration = stopwatch.Elapsed
            };
        }
    }

    /// <summary>
    /// 获取或设置缓存项（Cache-Aside模式）
    /// </summary>
    public async Task<CacheOperationResult<T>> GetOrSetAsync<T>(
        string key,
        Func<Task<T>> factory,
        TimeSpan? expiry = null,
        CacheOperationContext context = null)
    {
        if (factory == null)
            throw new ArgumentNullException(nameof(factory));

        var stopwatch = Stopwatch.StartNew();
        context ??= new CacheOperationContext { Key = key };

        try
        {
            // 如果不强制刷新，先尝试获取缓存
            if (!context.ForceRefresh)
            {
                var getCacheResult = await GetAsync<T>(key, context);
                if (getCacheResult.Success)
                {
                    return getCacheResult;
                }
            }

            // 使用分布式锁防止缓存击穿
            var lockKey = $"{key}:getorset_lock";
            var lockAcquired = false;

            if (_options.EnableL2Cache && _l2HealthStatus)
            {
                try
                {
                    lockAcquired = await _l2Cache.SetIfNotExistsAsync(lockKey, "locked", TimeSpan.FromMinutes(1));
                }
                catch (Exception ex)
                {
                    _logger.LogWarning(ex, "Failed to acquire distributed lock for key: {Key}", key);
                }
            }

            if (lockAcquired || !_options.EnableL2Cache || !_l2HealthStatus)
            {
                try
                {
                    // 再次检查缓存（双重检查锁定）
                    if (!context.ForceRefresh)
                    {
                        var doubleCheckResult = await GetAsync<T>(key, context);
                        if (doubleCheckResult.Success)
                        {
```

```
                    return doubleCheckResult;
                }
            }

            // 执行工厂方法
            _logger.LogDebug("Executing factory method for key: {Key}", key);
            var value = await factory();

            // 设置到所有缓存层级
            var setResult = await SetAsync(key, value, expiry, context);

            return new CacheOperationResult<T>
            {
                Value = value,
                Success = setResult.Success,
                HitLevel = CacheLevel.None, // 表示从数据源获取
                Duration = stopwatch.Elapsed
            };
        }
        finally
        {
            // 释放分布式锁
            if (lockAcquired)
            {
                try
                {
                    await _l2Cache.RemoveAsync(lockKey);
                }
                catch (Exception ex)
                {
                    _logger.LogWarning(ex, "Failed to release distributed lock for key: {Key}", key);
                }
            }
        }
    }
    else
    {
        // 等待锁释放并重试获取缓存
        await Task.Delay(Random.Shared.Next(50, 200)); // 随机退避
        var retryResult = await GetAsync<T>(key, context);

        if (retryResult.Success)
        {
            return retryResult;
        }

        // 降级：直接执行工厂方法
        _logger.LogWarning("Failed to acquire lock and cache miss for key: {Key}, executing factory method", key);
        var fallbackValue = await factory();

        // 尝试异步设置缓存
        _ = Task.Run(async () =>
        {
            try
            {
                await SetAsync(key, fallbackValue, expiry, context);
            }
            catch (Exception ex)
            {
                _logger.LogWarning(ex, "Failed to set cache in fallback scenario for key: {Key}", key);
            }
        });

        return new CacheOperationResult<T>
        {
            Value = fallbackValue,
            Success = true,
            HitLevel = CacheLevel.None,
            Duration = stopwatch.Elapsed
        };
    }
}
```

```
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error in GetOrSetAsync for key: {Key}", key);

            // 最终降级：直接执行工厂方法
            try
            {
                var fallbackValue = await factory();
                return new CacheOperationResult<T>
                {
                    Value = fallbackValue,
                    Success = true,
                    HitLevel = CacheLevel.None,
                    Duration = stopwatch.Elapsed,
                    Error = $"Cache operation failed, used fallback: {ex.Message}"
                };
            }
            catch (Exception factoryEx)
            {
                _logger.LogError(factoryEx, "Factory method also failed for key: {Key}", key);
                return new CacheOperationResult<T>
                {
                    Success = false,
                    Error = $"Both cache and factory failed: {ex.Message}, {factoryEx.Message}",
                    Duration = stopwatch.Elapsed
                };
            }
        }
    }

    /// <summary>
    /// 异步设置缓存项
    /// </summary>
    public async Task<CacheOperationResult<bool>> SetAsync<T>(
        string key,
        T value,
        TimeSpan? expiry = null,
        CacheOperationContext context = null)
    {
        var stopwatch = Stopwatch.StartNew();
        context ??= new CacheOperationContext { Key = key };

        var results = new List<bool>();
        var errors = new List<string>();

        try
        {
            // 记录更新时间（用于同步控制）
            _recentUpdates.Add(key, DateTime.UtcNow);

            // 根据一致性策略决定同步还是异步设置
            if (_options.ConsistencyStrategy == CacheConsistencyStrategy.StrongConsistency)
            {
                // 强一致性：同步设置所有层级
                await SetAllLevelsAsync();
            }
            else
            {
                // 最终一致性：异步设置非关键层级
                await SetCriticalLevelAsync();
                _ = Task.Run(SetNonCriticalLevelsAsync);
            }

            // 发送同步事件
            if (_options.EnableCacheSync)
            {
                var syncEvent = new CacheSyncEvent
                {
                    EventType = CacheSyncEventType.Set,
                    Key = key,
                    Timestamp = DateTime.UtcNow
                };
```

```
            _ = Task.Run(async () =>
            {
                try
                {
                    await _syncService.PublishAsync(syncEvent);
                }
                catch (Exception ex)
                {
                    _logger.LogWarning(ex, "Failed to publish sync event for key: {Key}", key);
                }
            });
        }

        var success = results.Count > 0 && results.Any(r => r);

        return new CacheOperationResult<bool>
        {
            Value = success,
            Success = success,
            Duration = stopwatch.Elapsed,
            Error = errors.Count > 0 ? string.Join("; ", errors) : null
        };
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error setting cache for key: {Key}", key);

        return new CacheOperationResult<bool>
        {
            Success = false,
            Error = ex.Message,
            Duration = stopwatch.Elapsed
        };
    }

    // 本地方法：设置所有层级（同步）
    async Task SetAllLevelsAsync()
    {
        var tasks = new List<Task<bool>>();

        if (_options.EnableL1Cache && (context.TargetLevel & CacheLevel.L1) != 0)
        {
            tasks.Add(SetL1CacheAsync());
        }

        if (_options.EnableL2Cache && (context.TargetLevel & CacheLevel.L2) != 0 && _l2HealthStatus)
        {
            tasks.Add(SetL2CacheAsync());
        }

        var taskResults = await Task.WhenAll(tasks);
        results.AddRange(taskResults);
    }

    // 本地方法：设置关键层级（通常是L1）
    async Task SetCriticalLevelAsync()
    {
        if (_options.EnableL1Cache && (context.TargetLevel & CacheLevel.L1) != 0)
        {
            var result = await SetL1CacheAsync();
            results.Add(result);
        }
    }

    // 本地方法：异步设置非关键层级（通常是L2）
    async Task SetNonCriticalLevelsAsync()
    {
        if (_options.EnableL2Cache && (context.TargetLevel & CacheLevel.L2) != 0 && _l2HealthStatus)
        {
            try
            {
```

```
                await Task.Delay(_options.L2WriteDelay); // 可选的写延迟
                await SetL2CacheAsync();
            }
            catch (Exception ex)
            {
                _logger.LogWarning(ex, "Failed to set L2 cache for key: {Key}", key);
            }
        }
    }

    // 本地方法：设置L1缓存
    async Task<bool> SetL1CacheAsync()
    {
        try
        {
            await _l1Cache.SetAsync(key, value, expiry ?? context.CustomExpiry);
            _logger.LogTrace("Set L1 cache for key: {Key}", key);
            return true;
        }
        catch (Exception ex)
        {
            var error = $"L1 set failed: {ex.Message}";
            errors.Add(error);
            _logger.LogWarning(ex, "Failed to set L1 cache for key: {Key}", key);
            return false;
        }
    }

    // 本地方法：设置L2缓存
    async Task<bool> SetL2CacheAsync()
    {
        try
        {
            await _l2Cache.SetAsync(key, value, expiry ?? context.CustomExpiry);
            _logger.LogTrace("Set L2 cache for key: {Key}", key);
            return true;
        }
        catch (Exception ex)
        {
            var error = $"L2 set failed: {ex.Message}";
            errors.Add(error);
            _logger.LogWarning(ex, "Failed to set L2 cache for key: {Key}", key);

            // L2缓存失败时标记不健康
            _l2HealthStatus = false;
            return false;
        }
    }
}

/// <summary>
/// 异步移除缓存项
/// </summary>
public async Task<CacheOperationResult<bool>> RemoveAsync(string key, CacheOperationContext context = null)
{
    var stopwatch = Stopwatch.StartNew();
    context ??= new CacheOperationContext { Key = key };

    var results = new List<bool>();
    var errors = new List<string>();

    try
    {
        // 并行移除所有层级
        var tasks = new List<Task<bool>>();

        if (_options.EnableL1Cache && (context.TargetLevel & CacheLevel.L1) != 0)
        {
            tasks.Add(RemoveL1CacheAsync());
        }

        if (_options.EnableL2Cache && (context.TargetLevel & CacheLevel.L2) != 0 && _l2HealthStatus)
```

```
        {
            tasks.Add(RemoveL2CacheAsync());
        }

        if (tasks.Count > 0)
        {
            var taskResults = await Task.WhenAll(tasks);
            results.AddRange(taskResults);
        }

        // 发送同步事件
        if (_options.EnableCacheSync)
        {
            var syncEvent = new CacheSyncEvent
            {
                EventType = CacheSyncEventType.Remove,
                Key = key,
                Timestamp = DateTime.UtcNow
            };

            _ = Task.Run(async () =>
            {
                try
                {
                    await _syncService.PublishAsync(syncEvent);
                }
                catch (Exception ex)
                {
                    _logger.LogWarning(ex, "Failed to publish sync event for key: {Key}", key);
                }
            });
        }

        // 清理更新记录
        _recentUpdates.Remove(key);

        var success = results.Count > 0 && results.Any(r => r);

        return new CacheOperationResult<bool>
        {
            Value = success,
            Success = true, // 移除操作总是被认为是成功的
            Duration = stopwatch.Elapsed,
            Error = errors.Count > 0 ? string.Join("; ", errors) : null
        };
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error removing cache for key: {Key}", key);

        return new CacheOperationResult<bool>
        {
            Success = false,
            Error = ex.Message,
            Duration = stopwatch.Elapsed
        };
    }

    // 本地方法：移除L1缓存
    async Task<bool> RemoveL1CacheAsync()
    {
        try
        {
            await _l1Cache.RemoveAsync(key);
            _logger.LogTrace("Removed L1 cache for key: {Key}", key);
            return true;
        }
        catch (Exception ex)
        {
            var error = $"L1 remove failed: {ex.Message}";
            errors.Add(error);
            _logger.LogWarning(ex, "Failed to remove L1 cache for key: {Key}", key);
```

```
                return false;
            }
        }

        // 本地方法：移除L2缓存
        async Task<bool> RemoveL2CacheAsync()
        {
            try
            {
                var result = await _l2Cache.RemoveAsync(key);
                _logger.LogTrace("Removed L2 cache for key: {Key}, success: {Success}", key, result);
                return result;
            }
            catch (Exception ex)
            {
                var error = $"L2 remove failed: {ex.Message}";
                errors.Add(error);
                _logger.LogWarning(ex, "Failed to remove L2 cache for key: {Key}", key);
                return false;
            }
        }
    }

    /// <summary>
    /// 根据模式批量移除缓存项
    /// </summary>
    public async Task<CacheOperationResult<long>> RemoveByPatternAsync(string pattern, CacheOperationContext context = null)
    {
        var stopwatch = Stopwatch.StartNew();
        context ??= new CacheOperationContext();

        long totalRemoved = 0;
        var errors = new List<string>();

        try
        {
            // L1缓存模式删除
            if (_options.EnableL1Cache && (context.TargetLevel & CacheLevel.L1) != 0)
            {
                try
                {
                    await _l1Cache.RemoveByPatternAsync(pattern);
                    _logger.LogDebug("Removed L1 cache entries by pattern: {Pattern}", pattern);
                }
                catch (Exception ex)
                {
                    errors.Add($"L1 pattern remove failed: {ex.Message}");
                    _logger.LogWarning(ex, "Failed to remove L1 cache by pattern: {Pattern}", pattern);
                }
            }

            // L2缓存模式删除
            if (_options.EnableL2Cache && (context.TargetLevel & CacheLevel.L2) != 0 && _l2HealthStatus)
            {
                try
                {
                    var removedCount = await _l2Cache.RemoveByPatternAsync(pattern);
                    totalRemoved += removedCount;
                    _logger.LogDebug("Removed {Count} L2 cache entries by pattern: {Pattern}", removedCount, pattern);
                }
                catch (Exception ex)
                {
                    errors.Add($"L2 pattern remove failed: {ex.Message}");
                    _logger.LogWarning(ex, "Failed to remove L2 cache by pattern: {Pattern}", pattern);
                }
            }

            // 发送同步事件
            if (_options.EnableCacheSync)
            {
                var syncEvent = new CacheSyncEvent
                {
```

```
                EventType = CacheSyncEventType.RemovePattern,
                Pattern = pattern,
                Timestamp = DateTime.UtcNow
            };

            _ = Task.Run(async () =>
            {
                try
                {
                    await _syncService.PublishAsync(syncEvent);
                }
                catch (Exception ex)
                {
                    _logger.LogWarning(ex, "Failed to publish sync event for pattern: {Pattern}", pattern);
                }
            });
        }

        return new CacheOperationResult<long>
        {
            Value = totalRemoved,
            Success = true,
            Duration = stopwatch.Elapsed,
            Error = errors.Count > 0 ? string.Join("; ", errors) : null
        };
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error removing cache by pattern: {Pattern}", pattern);

        return new CacheOperationResult<long>
        {
            Success = false,
            Error = ex.Message,
            Duration = stopwatch.Elapsed
        };
    }
}

/// <summary>
/// 检查缓存项是否存在
/// </summary>
public async Task<CacheOperationResult<bool>> ExistsAsync(string key, CacheLevel level = CacheLevel.All)
{
    var stopwatch = Stopwatch.StartNew();

    try
    {
        // 检查L1缓存
        if (_options.EnableL1Cache && (level & CacheLevel.L1) != 0)
        {
            var l1Result = await _l1Cache.GetAsync<object>(key);
            if (l1Result != null)
            {
                return new CacheOperationResult<bool>
                {
                    Value = true,
                    Success = true,
                    HitLevel = CacheLevel.L1,
                    Duration = stopwatch.Elapsed
                };
            }
        }

        // 检查L2缓存
        if (_options.EnableL2Cache && (level & CacheLevel.L2) != 0 && _l2HealthStatus)
        {
            var l2Exists = await _l2Cache.ExistsAsync(key);
            if (l2Exists)
            {
                return new CacheOperationResult<bool>
                {
```

```
                    Value = true,
                    Success = true,
                    HitLevel = CacheLevel.L2,
                    Duration = stopwatch.Elapsed
                };
            }
        }

        return new CacheOperationResult<bool>
        {
            Value = false,
            Success = true,
            HitLevel = CacheLevel.None,
            Duration = stopwatch.Elapsed
        };
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error checking cache existence for key: {Key}", key);

        return new CacheOperationResult<bool>
        {
            Success = false,
            Error = ex.Message,
            Duration = stopwatch.Elapsed
        };
    }
}

/// <summary>
/// 获取统计信息
/// </summary>
public async Task<MultiLevelCacheStatistics> GetStatisticsAsync()
{
    try
    {
        var l1Stats = _l1Cache.GetStatistics();
        var l2Stats = new CacheStatistics(); // Redis缓存统计需要自定义实现

        var totalOperations = Interlocked.Read(ref _totalOperations);
        var totalHits = Interlocked.Read(ref _l1Hits) + Interlocked.Read(ref _l2Hits);
        var totalMisses = Interlocked.Read(ref _totalMisses);

        return new MultiLevelCacheStatistics
        {
            L1Statistics = l1Stats,
            L2Statistics = l2Stats,
            TotalOperations = totalOperations,
            OverallHitRatio = totalOperations == 0 ? 0 : (double)totalHits / totalOperations,
            PerformanceMetrics = new Dictionary<string, object>
            {
                ["L1HitRatio"] = stats.L1HitRatio,
                ["L2HitRatio"] = stats.L2HitRatio,
                ["L2HealthStatus"] = _l2HealthStatus,
                ["IsDegraded"] = _isDegraded,
                ["DegradationDuration"] = _isDegraded ? DateTime.UtcNow - _degradationStartTime : TimeSpan.Zero
            }
        };
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error getting cache statistics");
        return new MultiLevelCacheStatistics();
    }
}

/// <summary>
/// 检查缓存服务健康状态
/// </summary>
public async Task<bool> IsHealthyAsync()
{
    try
```

```
        {
            var l1Healthy = true; // 内存缓存通常总是健康的
            var l2Healthy = true;
            var syncHealthy = true;

            // 检查L2缓存健康状态
            if (_options.EnableL2Cache)
            {
                try
                {
                    // 简单的ping测试
                    await _l2Cache.SetAsync("health_check", "ok", TimeSpan.FromSeconds(10));
                    l2Healthy = await _l2Cache.ExistsAsync("health_check");
                    await _l2Cache.RemoveAsync("health_check");
                }
                catch
                {
                    l2Healthy = false;
                }
            }

            // 检查同步服务健康状态
            if (_options.EnableCacheSync)
            {
                syncHealthy = _syncService.IsHealthy;
            }

            var overallHealthy = l1Healthy && (!_options.EnableL2Cache || l2Healthy) && (!_options.EnableCacheSync || syncHealt

            // 更新降级状态
            if (!overallHealthy && !_isDegraded)
            {
                _isDegraded = true;
                _degradationStartTime = DateTime.UtcNow;
                _logger.LogWarning("Cache service entered degraded mode");
            }
            else if (overallHealthy && _isDegraded)
            {
                _isDegraded = false;
                _logger.LogInformation("Cache service recovered from degraded mode after {Duration}",
                    DateTime.UtcNow - _degradationStartTime);
            }

            return overallHealthy;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error checking cache health");
            return false;
        }
    }

    /// <summary>
    /// 清空缓存
    /// </summary>
    public async Task ClearAsync(CacheLevel level = CacheLevel.All)
    {
        try
        {
            var tasks = new List<Task>();

            // 清空L1缓存（通过模式删除）
            if (_options.EnableL1Cache && (level & CacheLevel.L1) != 0)
            {
                tasks.Add(Task.Run(async () =>
                {
                    try
                    {
                        await _l1Cache.RemoveByPatternAsync("*");
                        _logger.LogInformation("Cleared L1 cache");
                    }
                    catch (Exception ex)
```

```
                {
                    _logger.LogError(ex, "Failed to clear L1 cache");
                }
            }));
        }

        // 清空L2缓存
        if (_options.EnableL2Cache && (level & CacheLevel.L2) != 0 && _l2HealthStatus)
        {
            tasks.Add(Task.Run(async () =>
            {
                try
                {
                    await _l2Cache.RemoveByPatternAsync("*");
                    _logger.LogInformation("Cleared L2 cache");
                }
                catch (Exception ex)
                {
                    _logger.LogError(ex, "Failed to clear L2 cache");
                }
            }));
        }

        await Task.WhenAll(tasks);

        // 发送清空同步事件
        if (_options.EnableCacheSync)
        {
            var syncEvent = new CacheSyncEvent
            {
                EventType = CacheSyncEventType.Clear,
                Timestamp = DateTime.UtcNow
            };

            _ = Task.Run(async () =>
            {
                try
                {
                    await _syncService.PublishAsync(syncEvent);
                }
                catch (Exception ex)
                {
                    _logger.LogWarning(ex, "Failed to publish clear sync event");
                }
            });
        }

        // 重置统计计数器
        _statisticsTracker.Reset();

        // 清空更新记录
        _recentUpdates.Clear();
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error clearing cache");
        throw;
    }
}

#region 私有方法

/// <summary>
/// 处理接收到的缓存同步事件
/// </summary>
/// <param name="syncEvent">同步事件</param>
private async Task OnCacheSyncEventReceived(CacheSyncEvent syncEvent)
{
    try
    {
        _logger.LogDebug("Received sync event: {EventType} for key: {Key}", syncEvent.EventType, syncEvent.Key);
```

```
            // 检查是否为最近的本地更新，避免循环同步
            if (!string.IsNullOrEmpty(syncEvent.Key) &&
                _recentUpdates.TryGet(syncEvent.Key, out var updateTime) &&
                (DateTime.UtcNow - updateTime).TotalSeconds < 5)
            {
                _logger.LogTrace("Skipping sync for recent local update: {Key}", syncEvent.Key);
                return;
            }

            switch (syncEvent.EventType)
            {
                case CacheSyncEventType.Remove:
                    await _l1Cache.RemoveAsync(syncEvent.Key);
                    break;

                case CacheSyncEventType.RemovePattern:
                    await _l1Cache.RemoveByPatternAsync(syncEvent.Pattern);
                    break;

                case CacheSyncEventType.Clear:
                    await _l1Cache.RemoveByPatternAsync("*");
                    break;

                case CacheSyncEventType.Expire:
                case CacheSyncEventType.Set:
                    // 对于设置操作，直接删除L1缓存项，让下次访问时从L2缓存重新加载
                    await _l1Cache.RemoveAsync(syncEvent.Key);
                    break;
            }

            _logger.LogTrace("Processed sync event: {EventType} for key: {Key}", syncEvent.EventType, syncEvent.Key);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error processing sync event: {EventType} for key: {Key}",
                syncEvent.EventType, syncEvent.Key);
        }
    }

    /// <summary>
    /// 执行定期健康检查
    /// </summary>
    /// <param name="state">定时器状态</param>
    private async void PerformHealthCheck(object state)
    {
        try
        {
            var previousL2Status = _l2HealthStatus;

            // 更新L2缓存健康状态
            if (_options.EnableL2Cache)
            {
                try
                {
                    var testKey = $"health_check_{Guid.NewGuid():N}";
                    await _l2Cache.SetAsync(testKey, "test", TimeSpan.FromSeconds(5));
                    _l2HealthStatus = await _l2Cache.ExistsAsync(testKey);
                    await _l2Cache.RemoveAsync(testKey);
                }
                catch (Exception ex)
                {
                    _l2HealthStatus = false;
                    _logger.LogWarning(ex, "L2 cache health check failed");
                }
            }

            // 记录状态变化
            if (previousL2Status != _l2HealthStatus)
            {
                if (_l2HealthStatus)
                {
                    _logger.LogInformation("L2 cache health recovered");
```

```
                }
                else
                {
                    _logger.LogWarning("L2 cache health degraded");
                }
            }

            // 清理过期的更新记录
            var cutoffTime = DateTime.UtcNow.AddMinutes(-5);
            var expiredKeys = _recentUpdates
                .Where(kvp => kvp.Value < cutoffTime)
                .Select(kvp => kvp.Key)
                .ToList();

            // LRU缓存已经处理了过期清理，这里不再需要手动操作
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error in health check");
        }
    }

    #endregion

    /// <summary>
    /// 释放资源
    /// </summary>
    public void Dispose()
    {
        if (!_disposed)
        {
            _healthCheckTimer?.Dispose();
            _syncService?.Dispose();
            _disposed = true;
        }
    }
}
```

7. 最佳实践和性能优化

7.1 缓存设计最佳实践

7.1.1 缓存键设计原则

分层命名规范:

```
// 推荐的键命名模式
public static class CacheKeyPatterns
{
    // 基础模式：{应用名}:{业务域}:{实体}:{标识符}
    public const string UserProfile = "myapp:user:profile:{0}";
    public const string UserPermissions = "myapp:user:permissions:{0}";
    public const string ProductList = "myapp:product:list:page:{0}:size:{1}";

    // 会话相关：{应用名}:{业务域}:session:{会话ID}:{实体}
    public const string UserSession = "myapp:user:session:{0}:settings";

    // 临时数据：{应用名}:temp:{业务场景}:{标识符}
    public const string TempData = "myapp:temp:upload:{0}";

    // 配置数据：{应用名}:config:{配置类型}
    public const string SystemConfig = "myapp:config:system";

    // 统计数据：{应用名}:stats:{时间维度}:{标识符}
    public const string DailyStats = "myapp:stats:daily:{0:yyyyMMdd}";
}

/// <summary>
/// 缓存键构建器
/// </summary>
```

```
public class CacheKeyBuilder
{
    private readonly string _applicationName;
    private readonly List<string> _segments;

    public CacheKeyBuilder(string applicationName)
    {
        _applicationName = applicationName ?? throw new ArgumentNullException(nameof(applicationName));
        _segments = new List<string> { _applicationName };
    }

    public CacheKeyBuilder Domain(string domain)
    {
        _segments.Add(domain);
        return this;
    }

    public CacheKeyBuilder Entity(string entity)
    {
        _segments.Add(entity);
        return this;
    }

    public CacheKeyBuilder Id(object id)
    {
        _segments.Add(id?.ToString() ?? "null");
        return this;
    }

    public CacheKeyBuilder Attribute(string attribute)
    {
        _segments.Add(attribute);
        return this;
    }

    public CacheKeyBuilder Session(string sessionId)
    {
        _segments.Add("session");
        _segments.Add(sessionId);
        return this;
    }

    public CacheKeyBuilder WithParameters(params object[] parameters)
    {
        foreach (var param in parameters)
        {
            _segments.Add(param?.ToString() ?? "null");
        }
        return this;
    }

    public string Build()
    {
        return string.Join(":", _segments);
    }

    public override string ToString() => Build();
}
```

7.1.2 过期策略优化

智能过期时间计算:

```
/// <summary>
/// 智能过期策略
/// </summary>
public class SmartExpirationStrategy
{
    private readonly ILogger<SmartExpirationStrategy> _logger;
    private readonly Random _random = new();
```

```
    public SmartExpirationStrategy(ILogger<SmartExpirationStrategy> logger)
    {
        _logger = logger;
    }

    /// <summary>
    /// 根据数据类型和访问模式计算过期时间
    /// </summary>
    /// <param name="dataType">数据类型</param>
    /// <param name="accessFrequency">访问频率</param>
    /// <param name="dataVolatility">数据变化频率</param>
    /// <param name="businessCritical">是否业务关键</param>
    /// <returns>推荐的过期时间</returns>
    public TimeSpan CalculateExpiry(
        CacheDataType dataType,
        AccessFrequency accessFrequency,
        DataVolatility dataVolatility,
        bool businessCritical = false)
    {
        // 基础过期时间
        var baseExpiry = dataType switch
        {
            CacheDataType.UserProfile => TimeSpan.FromHours(4),
            CacheDataType.SystemConfiguration => TimeSpan.FromHours(12),
            CacheDataType.ProductCatalog => TimeSpan.FromHours(2),
            CacheDataType.UserPermissions => TimeSpan.FromHours(1),
            CacheDataType.SessionData => TimeSpan.FromMinutes(30),
            CacheDataType.TemporaryData => TimeSpan.FromMinutes(5),
            CacheDataType.StatisticsData => TimeSpan.FromMinutes(15),
            _ => TimeSpan.FromHours(1)
        };

        // 根据访问频率调整
        var frequencyMultiplier = accessFrequency switch
        {
            AccessFrequency.VeryHigh => 2.0,
            AccessFrequency.High => 1.5,
            AccessFrequency.Medium => 1.0,
            AccessFrequency.Low => 0.7,
            AccessFrequency.VeryLow => 0.5,
            _ => 1.0
        };

        // 根据数据变化频率调整
        var volatilityMultiplier = dataVolatility switch
        {
            DataVolatility.VeryHigh => 0.3,
            DataVolatility.High => 0.5,
            DataVolatility.Medium => 0.8,
            DataVolatility.Low => 1.2,
            DataVolatility.VeryLow => 1.5,
            _ => 1.0
        };

        // 业务关键数据缩短过期时间以确保一致性
        var criticalMultiplier = businessCritical ? 0.8 : 1.0;

        // 计算最终过期时间
        var finalExpiry = TimeSpan.FromMilliseconds(
            baseExpiry.TotalMilliseconds *
            frequencyMultiplier *
            volatilityMultiplier *
            criticalMultiplier);

        // 添加随机偏移防止缓存雪崩（±10%）
        var jitterPercentage = _random.NextDouble() * 0.2 - 0.1; // -10% to +10%
        finalExpiry = TimeSpan.FromMilliseconds(
            finalExpiry.TotalMilliseconds * (1 + jitterPercentage));

        // 确保最小和最大边界
        var minExpiry = TimeSpan.FromMinutes(1);
        var maxExpiry = TimeSpan.FromDays(1);
```

```
        if (finalExpiry < minExpiry) finalExpiry = minExpiry;
        if (finalExpiry > maxExpiry) finalExpiry = maxExpiry;

        _logger.LogDebug("Calculated expiry for {DataType}: {Expiry} " +
            "(base: {BaseExpiry}, freq: {FreqMultiplier:F1}x, vol: {VolMultiplier:F1}x, critical: {CriticalMultiplier:F1}x)",
            dataType, finalExpiry, baseExpiry, frequencyMultiplier, volatilityMultiplier, criticalMultiplier);

        return finalExpiry;
    }
}

public enum CacheDataType
{
    UserProfile,
    SystemConfiguration,
    ProductCatalog,
    UserPermissions,
    SessionData,
    TemporaryData,
    StatisticsData
}

public enum AccessFrequency
{
    VeryLow,
    Low,
    Medium,
    High,
    VeryHigh
}

public enum DataVolatility
{
    VeryLow,    // 几乎不变化，如系统配置
    Low,        // 很少变化，如用户档案
    Medium,     // 定期变化，如产品信息
    High,       // 频繁变化，如库存数据
    VeryHigh    // 实时变化，如在线用户状态
}
```

7.1.3 缓存预热策略

```
/// <summary>
/// 缓存预热服务
/// </summary>
public interface ICacheWarmupService
{
    Task WarmupAsync(CancellationToken cancellationToken = default);
    Task WarmupSpecificDataAsync(string dataType, CancellationToken cancellationToken = default);
}

/// <summary>
/// 缓存预热服务实现
/// </summary>
public class CacheWarmupService : ICacheWarmupService
{
    private readonly IMultiLevelCacheManager _cacheManager;
    private readonly IServiceProvider _serviceProvider;
    private readonly ILogger<CacheWarmupService> _logger;
    private readonly CacheWarmupOptions _options;

    public CacheWarmupService(
        IMultiLevelCacheManager cacheManager,
        IServiceProvider serviceProvider,
        ILogger<CacheWarmupService> logger,
        IOptions<CacheWarmupOptions> options)
    {
        _cacheManager = cacheManager;
        _serviceProvider = serviceProvider;
        _logger = logger;
```

```
        _options = options.Value;
    }

    /// <summary>
    /// 执行完整的缓存预热
    /// </summary>
    public async Task WarmupAsync(CancellationToken cancellationToken = default)
    {
        _logger.LogInformation("Starting cache warmup process");
        var stopwatch = Stopwatch.StartNew();

        try
        {
            var warmupTasks = new List<Task>
            {
                WarmupSystemConfigurationAsync(cancellationToken),
                WarmupHotUserDataAsync(cancellationToken),
                WarmupProductCatalogAsync(cancellationToken),
                WarmupFrequentlyAccessedDataAsync(cancellationToken)
            };

            await Task.WhenAll(warmupTasks);

            _logger.LogInformation("Cache warmup completed in {Duration:F2}s", stopwatch.Elapsed.TotalSeconds);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Cache warmup failed after {Duration:F2}s", stopwatch.Elapsed.TotalSeconds);
            throw;
        }
    }

    /// <summary>
    /// 预热系统配置数据
    /// </summary>
    private async Task WarmupSystemConfigurationAsync(CancellationToken cancellationToken)
    {
        try
        {
            _logger.LogDebug("Warming up system configuration");

            using var scope = _serviceProvider.CreateScope();
            var configService = scope.ServiceProvider.GetRequiredService<IConfigurationService>();

            var configKeys = new[]
            {
                "app_settings",
                "feature_flags",
                "business_rules",
                "system_parameters"
            };

            var tasks = configKeys.Select(async key =>
            {
                var cacheKey = new CacheKeyBuilder("myapp")
                    .Domain("config")
                    .Entity(key)
                    .Build();

                await _cacheManager.GetOrSetAsync(
                    cacheKey,
                    async () => await configService.GetConfigurationAsync(key),
                    TimeSpan.FromHours(12));
            });

            await Task.WhenAll(tasks);
            _logger.LogDebug("System configuration warmup completed");
        }
        catch (Exception ex)
        {
            _logger.LogWarning(ex, "Failed to warmup system configuration");
        }
```

```
    }

    /// <summary>
    /// 预热热点用户数据
    /// </summary>
    private async Task WarmupHotUserDataAsync(CancellationToken cancellationToken)
    {
        try
        {
            _logger.LogDebug("Warming up hot user data");

            using var scope = _serviceProvider.CreateScope();
            var userService = scope.ServiceProvider.GetRequiredService<IUserService>();
            var analyticsService = scope.ServiceProvider.GetRequiredService<IAnalyticsService>();

            // 获取最近活跃用户列表
            var activeUserIds = await analyticsService.GetRecentlyActiveUsersAsync(
                TimeSpan.FromDays(7),
                _options.TopUsersToWarmup);

            var semaphore = new SemaphoreSlim(_options.MaxConcurrency);
            var tasks = activeUserIds.Select(async userId =>
            {
                await semaphore.WaitAsync(cancellationToken);
                try
                {
                    // 预热用户基本信息
                    var userCacheKey = new CacheKeyBuilder("myapp")
                        .Domain("user")
                        .Entity("profile")
                        .Id(userId)
                        .Build();

                    await _cacheManager.GetOrSetAsync(
                        userCacheKey,
                        async () => await userService.GetUserByIdAsync(userId),
                        TimeSpan.FromHours(4));

                    // 预热用户权限
                    var permissionsCacheKey = new CacheKeyBuilder("myapp")
                        .Domain("user")
                        .Entity("permissions")
                        .Id(userId)
                        .Build();

                    await _cacheManager.GetOrSetAsync(
                        permissionsCacheKey,
                        async () => await userService.GetUserPermissionsAsync(userId),
                        TimeSpan.FromHours(2));
                }
                finally
                {
                    semaphore.Release();
                }
            });

            await Task.WhenAll(tasks);
            _logger.LogDebug("Hot user data warmup completed for {Count} users", activeUserIds.Count);
        }
        catch (Exception ex)
        {
            _logger.LogWarning(ex, "Failed to warmup hot user data");
        }
    }

    /// <summary>
    /// 预热产品目录数据
    /// </summary>
    private async Task WarmupProductCatalogAsync(CancellationToken cancellationToken)
    {
        try
        {
```

```
            _logger.LogDebug("Warming up product catalog");

            using var scope = _serviceProvider.CreateScope();
            var productService = scope.ServiceProvider.GetRequiredService<IProductService>();

            // 预热热门产品分类
            var popularCategories = await productService.GetPopularCategoriesAsync(_options.TopCategoriesToWarmup);

            var tasks = popularCategories.Select(async category =>
            {
                var cacheKey = new CacheKeyBuilder("myapp")
                    .Domain("product")
                    .Entity("category")
                    .Id(category.Id)
                    .Build();

                await _cacheManager.GetOrSetAsync(
                    cacheKey,
                    async () => await productService.GetCategoryProductsAsync(category.Id, 1, 20),
                    TimeSpan.FromHours(2));
            });

            await Task.WhenAll(tasks);
            _logger.LogDebug("Product catalog warmup completed for {Count} categories", popularCategories.Count);
        }
        catch (Exception ex)
        {
            _logger.LogWarning(ex, "Failed to warmup product catalog");
        }
    }

    /// <summary>
    /// 预热频繁访问的数据
    /// </summary>
    private async Task WarmupFrequentlyAccessedDataAsync(CancellationToken cancellationToken)
    {
        try
        {
            _logger.LogDebug("Warming up frequently accessed data");

            // 这里可以根据实际的访问日志或分析数据来确定需要预热的内容
            // 示例：预热首页数据、热门搜索结果等

            var commonQueries = new[]
            {
                ("homepage_banner", TimeSpan.FromHours(6)),
                ("popular_products", TimeSpan.FromHours(1)),
                ("trending_categories", TimeSpan.FromMinutes(30)),
                ("system_announcements", TimeSpan.FromHours(4))
            };

            using var scope = _serviceProvider.CreateScope();
            var contentService = scope.ServiceProvider.GetRequiredService<IContentService>();

            var tasks = commonQueries.Select(async query =>
            {
                var (queryType, expiry) = query;
                var cacheKey = new CacheKeyBuilder("myapp")
                    .Domain("content")
                    .Entity(queryType)
                    .Build();

                await _cacheManager.GetOrSetAsync(
                    cacheKey,
                    async () => await contentService.GetContentAsync(queryType),
                    expiry);
            });

            await Task.WhenAll(tasks);
            _logger.LogDebug("Frequently accessed data warmup completed");
        }
        catch (Exception ex)
```

```
        {
            _logger.LogWarning(ex, "Failed to warmup frequently accessed data");
        }
    }

    /// <summary>
    /// 预热特定类型的数据
    /// </summary>
    public async Task WarmupSpecificDataAsync(string dataType, CancellationToken cancellationToken = default)
    {
        _logger.LogInformation("Starting specific cache warmup for data type: {DataType}", dataType);

        try
        {
            switch (dataType.ToLowerInvariant())
            {
                case "config":
                case "configuration":
                    await WarmupSystemConfigurationAsync(cancellationToken);
                    break;

                case "user":
                case "users":
                    await WarmupHotUserDataAsync(cancellationToken);
                    break;

                case "product":
                case "products":
                    await WarmupProductCatalogAsync(cancellationToken);
                    break;

                case "content":
                    await WarmupFrequentlyAccessedDataAsync(cancellationToken);
                    break;

                default:
                    _logger.LogWarning("Unknown data type for warmup: {DataType}", dataType);
                    break;
            }

            _logger.LogInformation("Specific cache warmup completed for data type: {DataType}", dataType);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Specific cache warmup failed for data type: {DataType}", dataType);
            throw;
        }
    }
}

/// <summary>
/// 缓存预热配置选项
/// </summary>
public class CacheWarmupOptions
{
    public int TopUsersToWarmup { get; set; } = 1000;
    public int TopCategoriesToWarmup { get; set; } = 50;
    public int MaxConcurrency { get; set; } = Environment.ProcessorCount * 2;
    public bool EnableScheduledWarmup { get; set; } = true;
    public TimeSpan WarmupInterval { get; set; } = TimeSpan.FromHours(6);
    public List<string> WarmupDataTypes { get; set; } = new() { "config", "users", "products", "content" };
}
```

7.4 安全性和可靠性增强

基于深度技术分析的结果，我们对原有架构进行了重要的安全性和可靠性改进：

7.4.1 增强的异常处理机制

我们引入了分层的异常处理体系，将不同类型的缓存异常进行分类处理：

```
// 细分异常类型，提供更精确的错误处理
public class CacheConnectionException : CacheException { }
public class CacheSerializationException : CacheException { }
public class CacheTimeoutException : CacheException { }
public class CacheValidationException : CacheException { }

// 在缓存操作中使用分层异常处理
try
{
    var result = await factory();
    return result;
}
catch (CacheConnectionException ex)
{
    _logger.LogWarning(ex, "Cache connection failed, using fallback");
    return await factory(); // 优雅降级
}
catch (CacheSerializationException ex)
{
    _logger.LogError(ex, "Serialization failed");
    throw; // 序列化错误需要立即处理
}
```

7.4.2 线程安全的统计系统

原有的统计计数器存在线程安全问题，我们引入了专门的统计追踪器：

```
public class CacheStatisticsTracker
{
    private long _totalOperations = 0;
    private long _l1Hits = 0;
    private long _l2Hits = 0;
    private long _totalMisses = 0;

    public void RecordOperation() => Interlocked.Increment(ref _totalOperations);
    public void RecordHit(CacheLevel level) { /* 原子操作 */ }
    public CacheStatisticsSnapshot GetSnapshot() { /* 线程安全的快照 */ }
}
```

7.4.3 缓存数据验证和安全机制

为防止缓存投毒和数据安全问题，我们实现了多层验证机制：

```
public class DefaultCacheDataValidator : ICacheDataValidator
{
    public bool IsValid<T>(T value)
    {
        // 检查禁止类型
        if (_forbiddenTypes.Contains(value.GetType()))
            return false;

        // 检查循环引用
        if (HasCircularReference(value))
            return false;

        return true;
    }

    public void ValidateKey(string key)
    {
        // 验证key格式和长度
        if (!_keyValidationRegex.IsMatch(key))
            throw new CacheValidationException($"Invalid key: {key}");
    }
}
```

7.4.4 智能序列化性能优化

引入多种序列化器支持，根据数据类型自动选择最佳序列化方案：

```
public class SmartCacheSerializer : ICacheSerializer
{
    private readonly ICacheSerializer[] _serializers = new[]
    {
        new BinaryCacheSerializer(), // 优先使用高性能二进制序列化
        new JsonCacheSerializer()    // 备选JSON序列化
    };

    public byte[] Serialize<T>(T value)
    {
        foreach (var serializer in _serializers)
        {
            if (serializer.SupportsType(typeof(T)))
            {
                return serializer.Serialize(value);
            }
        }
        throw new CacheSerializationException("No suitable serializer found");
    }
}
```

7.4.5 断路器模式实现

实现断路器模式来处理Redis连接故障，提高系统的整体可靠性：

```
public class CacheCircuitBreaker
{
    private CircuitBreakerState _state = CircuitBreakerState.Closed;

    public async Task<T> ExecuteAsync<T>(Func<Task<T>> operation)
    {
        if (!CanExecute())
        {
            throw new CacheException("Circuit breaker is OPEN");
        }

        try
        {
            var result = await operation();
            OnSuccess();
            return result;
        }
        catch (Exception ex)
        {
            OnFailure(ex);
            throw;
        }
    }
}
```

7.4.6 LRU内存管理

为防止内存泄漏，我们用LRU缓存替换了原有的ConcurrentDictionary：

```
public class LRUCache<TKey, TValue>
{
    private readonly int _maxSize;
    private readonly Dictionary<TKey, LinkedListNode<CacheItem<TKey, TValue>>> _cache;
    private readonly LinkedList<CacheItem<TKey, TValue>> _lruList;

    public void Add(TKey key, TValue value)
    {
        // 检查容量限制
        if (_cache.Count >= _maxSize)
```

```
        {
            // 移除最久未使用的项
            var lastNode = _lruList.Last;
            _cache.Remove(lastNode.Value.Key);
            _lruList.RemoveLast();
        }

        // 添加新项到链表头部
        var newNode = _lruList.AddFirst(new CacheItem<TKey, TValue> { Key = key, Value = value });
        _cache[key] = newNode;
    }
}
```

8.1 学习资源和参考文献

8.1.1 官方文档

- Microsoft.Extensions.Caching.Memory
- StackExchange.Redis Documentation
- Redis Official Documentation

8.1.2 推荐书籍

- 《高性能MySQL》- 缓存设计理论基础
- 《Redis设计与实现》- Redis深度解析
- 《.NET性能优化》- .NET平台性能调优

8.1.3 开源项目参考

- EasyCaching - .NET缓存框架
- FusionCache - 高级缓存库
- CacheManager - 多级缓存管理器

结尾

qq技术交流群：737776595

合集: DotNet

239573049
粉丝 - 7 关注 - 0

« 上一篇： 震撼！CloseAI终于变回OpenAI了！GPT-OSS来了，这次真的不一样了🔥

posted on 2025-08-12 19:22 239573049 阅读(1041) 评论(6) 收藏 举报

相关博文：
· 免费包白嫖最新DeepSeek-V3驱动的MCP与SemanticKernel实...
· 使用.NET实现自带思考的Tool 并且提供mcp streamable http服...
· 乘风破浪，遇见最佳跨平台跨终端框架.Net Core/.Net生态 - 浅...
· 分布式缓存接口正确用法
· Redis笔记

阅读排行：
· .NET周刊【8月第1期 2025-08-03】
· Manus快速搭建个人网站
· 这套 Java 监控系统太香了！我连夜给项目加上了
· 【译】GPT-5 现已在 Visual Studio 中可用
· 千亿消息"过眼云烟"？Kafka把硬盘当内存用的性能魔法，全靠这一手！

公告

昵称： 239573049
园龄： 8个月
粉丝： 7
关注： 0
+加关注

< 2025年8月

日	一	二	三	四	五
27	28	29	30	31	1
3	4	5	6	7	8
10	11	12	13	14	15
17	18	19	20	21	22
24	25	26	27	28	29
31	1	2	3	4	5

搜索

常用链接

- 我的随笔
- 我的评论
- 我的参与
- 最新评论
- 我的标签

合集

MCP(3)
提示词(2)
DotNet(3)

随笔分类

DotNet(1)

随笔档案

2025年8月(2)
2025年7月(5)
2025年6月(4)
2025年5月(5)
2025年4月(5)
2025年3月(2)
2025年2月(1)

阅读排行榜

- 1. C#源生成器：让你的代码飞起来的黑科技(1847)
- 2. 使用.NET实现自带思考的Tool 并且提供mcp streamable http服务(1166)
- 3. C#中的多级缓存架构设计与实现深度解析(1041)
- 4. KoalaWiki vs DeepWiki：更优秀的开源代码知识库解决方案(1027)
- 5. （包含5w字提示词开源）手把手教你搭建开源提示词优化平台(948)

评论排行榜

- 1. （包含5w字提示词开源）手把手教你搭建开源提示词优化平台(11)
- 2. C#中的多级缓存架构设计与实现深度解析(6)
- 3. 使用Semantic Kernel实现Claude Code的Agents TODO能力(2)
- 4. 开源的DeekWiki加入MCP，为您的Cursor提供开源项目分析，轻松让AI掌握开源项目使用文档！(2)
- 5. 震撼！CloseAI终于变回OpenAI了！GPT-OSS来了，这次真的不一样了🔥(1)

推荐排行榜

- 1. C#源生成器：让你的代码飞起来的黑科技(14)
- 2. C#中的多级缓存架构设计与实现深度解析(10)
- 3. 最新DeepSeek-V3驱动的MCP与SemanticKernel实战教程 - 打造智能应用的终极指南(5)
- 4. 使用.NET实现自带思考的Tool 并且提供mcp streamable http服务(4)
- 5. 使用离线部署32B模型实现OpenDeepWiki项目代码自动分析与文档生成(4)

最新评论

- 1. Re:C#中的多级缓存架构设计与实现深度解析
- 这代码确实需要折叠一下。
-
- 2. Re:C#中的多级缓存架构设计与实现深度解析
- 太长不看系列.....
-
- 3. Re:C#中的多级缓存架构设计与实现深度解析
- 所以这个L1L2L3和CPU的三级缓存有啥关系.....代码还不折叠
-
- 4. Re:C#中的多级缓存架构设计与实现深度解析
- 加个代码折叠吧,看的难受
-
- 5. Re:C#中的多级缓存架构设计与实现深度解析
- 加个代码折叠吧,看的难受
-