In [1]:
```
# uv pip install lightgbm --python 3.12
```

In [2]:
```
# uv pip install optuna --python 3.12
```

임포트 분리

In [3]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from lightgbm import LGBMClassifier
# import shap
from xgboost import XGBClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import (
    classification_report,
    roc_auc_score,
    accuracy_score,
    precision_score,
    recall_score,
    f1_score
)

# 한글 폰트 설정 (Windows)
plt.rcParams['font.family'] = 'Malgun Gothic'
plt.rcParams['axes.unicode_minus'] = False

# 기록장 초기화
experiment_log = pd.DataFrame(columns=['Model_Name', 'Scale_Pos_Weight', 'Thresh
```

In [4]:
```
coc_df = pd.read_csv('C:/Users/tw2ps/Desktop/sparta_python/ML-Team_Project/coc_c
```

In [5]:
```
coc_df_clean = coc_df.copy()
```

In [6]:
```
columns_to_drop = [
    'clan_name',
    'clan_description',
    'clan_location',
    'clan_badge_url'
]

coc_df_clean = coc_df_clean.drop(columns=columns_to_drop, errors='ignore')

print(f"제거된 컬럼: {columns_to_drop}")
print(f"남은 컬럼 수: {len(coc_df_clean.columns)}개")
```

```
제거된 컬럼: ['clan_name', 'clan_description', 'clan_location', 'clan_badge_url']
남은 컬럼 수: 23개
```

In [7]:
```
#총 전쟁 횟수
coc_df_clean['war_total'] = (
```

```
    coc_df_clean['war_wins'] +
    coc_df_clean['war_ties'] +
    coc_df_clean['war_losses']
)

#승률
coc_df_clean['win_rate'] = coc_df_clean.apply(
    lambda row: row['war_wins'] / row['war_total'] if row['war_total'] > 0 else
    axis=1
)

#확인
print("파생변수 생성 완료:")
coc_df_clean[['clan_tag', 'war_wins', 'war_total', 'win_rate']].head()
```

파생변수 생성 완료:

Out[7]:

	clan_tag	war_wins	war_total	win_rate
0	#UQVQRJQ0	93	124	0.750000
1	#2QC9Y0CQU	0	0	0.000000
2	#202CJRP2U	1	1	1.000000
3	#2Y89RRGLY	7	18	0.388889
4	#99PU9QPY	3	14	0.214286

In [8]:
```
# 두 컬럼 비교
comparison_result = (coc_df_clean['required_builder_base_trophies'] == coc_df_cl

print(f"완전히 동일한가요?: {comparison_result}")

# 차이가 있다면 얼마나 다른지 확인 (샘플 보기)
if not comparison_result:
    diff_df = coc_df_clean[coc_df_clean['clan_builder_base_points'] != coc_df_cl
    print(f"\n차이가 있는 행 개수: {len(diff_df)}개")
    print(diff_df[['clan_builder_base_points', 'clan_versus_points']].head())
else:
    print("\n두 컬럼은 완벽하게 일치합니다. 하나는 삭제해도 됩니다.")
```

```
완전히 동일한가요?: True

두 컬럼은 완벽하게 일치합니다. 하나는 삭제해도 됩니다.
```

In [9]:
```
comparison_result = (coc_df_clean['clan_builder_base_points'] == coc_df_clean['c

print(f"완전히 동일한가요?: {comparison_result}")

# 차이가 있다면 얼마나 다른지 확인 (샘플 보기)
if not comparison_result:
    diff_df = coc_df_clean[coc_df_clean['clan_builder_base_points'] != coc_df_cl
    print(f"\n차이가 있는 행 개수: {len(diff_df)}개")
    print(diff_df[['clan_builder_base_points', 'clan_versus_points']].head())
else:
    print("\n두 컬럼은 완벽하게 일치합니다. 하나는 삭제해도 됩니다.")
```

```
완전히 동일한가요?: True

두 컬럼은 완벽하게 일치합니다. 하나는 삭제해도 됩니다.
```

In [10]:
```
# 비활성 클랜 조건 정의
# 조건 1: 멤버 5명 미만 (클랜전 최소 인원 미달)
# 조건 2: 레벨 2 이상인데 캐피탈 점수가 0 (과거엔 활동했으나 현재 중단)
# 조건 3: 전쟁 경험이 0

def is_ghost(row):
    # 멤버 5명 미만 -> 전쟁 자체가 불가능함
    if row['num_members'] < 5:
        return True
    # 레벨은 2 이상인데 캐피탈 점수가 0 → 활동 중단
    if row['clan_level'] >= 2 and row['clan_capital_points'] == 0:
        return True
    # 전쟁 경험 없음 → 분석 불가
    if row['war_total'] == 0:
        return True
    return False

# 비활성 클랜 여부 컬럼 생성
coc_df_clean['is_ghost'] = coc_df_clean.apply(is_ghost, axis=1)

# 비활성 클랜 비율 확인
ghost_count = coc_df_clean['is_ghost'].sum()
total_count = len(coc_df_clean)
print(f"유령 클랜 수: {ghost_count:,}개 ({ghost_count/total_count*100:.1f}%)")
print(f"정상 클랜 수: {total_count - ghost_count:,}개")

# 정상 클랜만 남기기 (is_ghost == False)
coc_df_active = coc_df_clean[coc_df_clean['is_ghost'] == False].copy()

print(f"\n분석용 데이터(coc_df_active) 생성 완료: {len(coc_df_active):,}개 클랜")
```

```
유령 클랜 수: 3,222,737개 (90.5%)
정상 클랜 수: 337,006개

분석용 데이터(coc_df_active) 생성 완료: 337,006개 클랜
```

In [11]:
```
# isFamilyFriendly: True -> 1, False -> 0
coc_df_active['isFamilyFriendly'] = coc_df_active['isFamilyFriendly'].astype(int

# 확인
print("인코딩 완료!")
print(coc_df_active['isFamilyFriendly'].value_counts())
```

```
인코딩 완료!
isFamilyFriendly
0    197299
1    139707
Name: count, dtype: int64
```

In [12]:
```
# 최종 데이터 구조 확인
print("=" * 50)
print("전처리 완료 데이터 (coc_df_active) 정보")
print("=" * 50)
coc_df_active.info()
```

```
==================================================
전처리 완료 데이터 (coc_df_active) 정보
==================================================
<class 'pandas.core.frame.DataFrame'>
Index: 337006 entries, 11 to 3559705
Data columns (total 26 columns):
 #   Column                          Non-Null Count   Dtype
---  ------                          --------------   -----
 0   clan_tag                        337006 non-null  object
 1   clan_type                       337006 non-null  object
 2   isFamilyFriendly                337006 non-null  int64
 3   clan_level                      337006 non-null  int64
 4   clan_points                     337006 non-null  int64
 5   clan_builder_base_points        337006 non-null  int64
 6   clan_versus_points              337006 non-null  int64
 7   required_trophies               337006 non-null  int64
 8   war_frequency                   337006 non-null  object
 9   war_win_streak                  337006 non-null  int64
 10  war_wins                        337006 non-null  int64
 11  war_ties                        337006 non-null  int64
 12  war_losses                      337006 non-null  int64
 13  clan_war_league                 337006 non-null  object
 14  num_members                     337006 non-null  int64
 15  required_builder_base_trophies  337006 non-null  int64
 16  required_versus_trophies        337006 non-null  int64
 17  required_townhall_level         337006 non-null  int64
 18  clan_capital_hall_level         337006 non-null  int64
 19  clan_capital_points             337006 non-null  int64
 20  capital_league                  337006 non-null  object
 21  mean_member_level               337006 non-null  int64
 22  mean_member_trophies            337006 non-null  int64
 23  war_total                       337006 non-null  int64
 24  win_rate                        337006 non-null  float64
 25  is_ghost                        337006 non-null  bool
dtypes: bool(1), float64(1), int64(19), object(5)
memory usage: 67.2+ MB
```

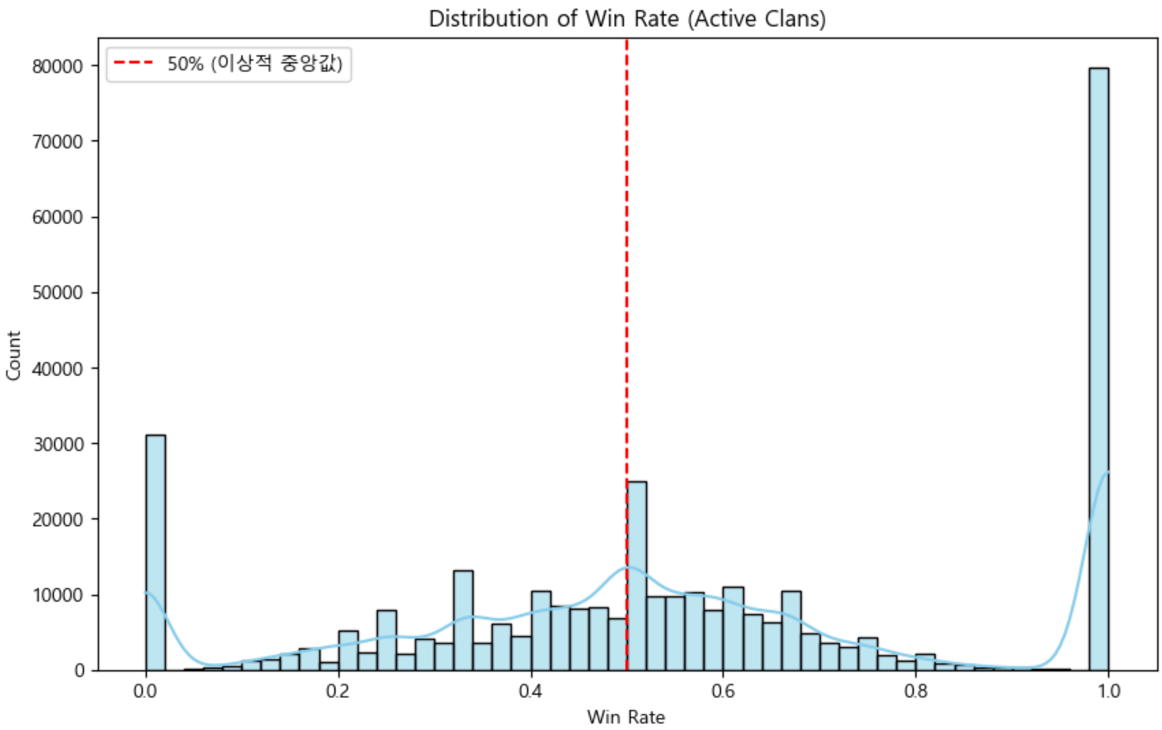
In [13]:
```python
# 수치형 데이터 기초 통계
coc_df_active.describe()
```

Out[13]:

	isFamilyFriendly	clan_level	clan_points	clan_builder_base_points	clan_ve
count	337006.000000	337006.000000	337006.000000	337006.000000	33
mean	0.414553	7.437992	14932.318205	14927.407746	1
std	0.492646	7.023098	12591.352375	12097.967308	1
min	0.000000	1.000000	0.000000	0.000000	
25%	0.000000	1.000000	3399.000000	3548.000000	
50%	0.000000	5.000000	11728.000000	12881.000000	1
75%	1.000000	12.000000	24067.000000	24267.750000	2
max	1.000000	36.000000	54721.000000	51686.000000	5

In [14]:
```python
# 승률 분포 시각화
plt.figure(figsize=(10, 6))
```

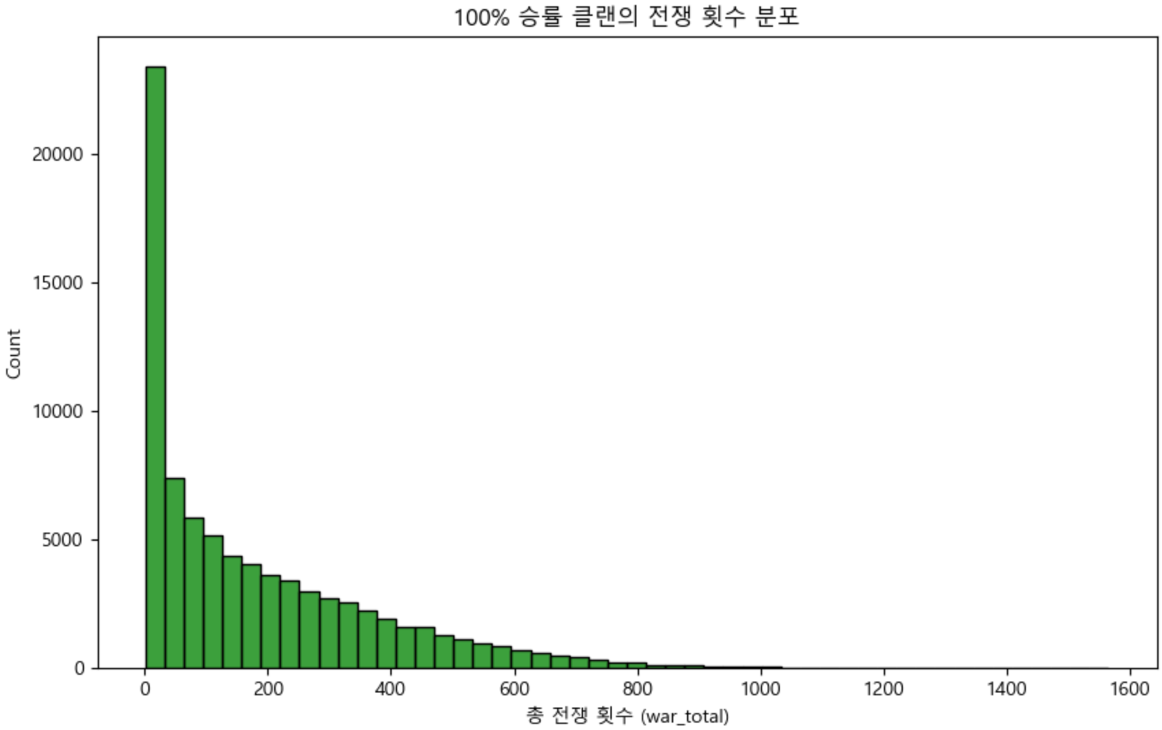
```
sns.histplot(coc_df_active['win_rate'], bins=50, kde=True, color='skyblue')
plt.axvline(x=0.5, color='red', linestyle='--', label='50% (이상적 중앙값)')
plt.title('Distribution of Win Rate (Active Clans)')
plt.xlabel('Win Rate')
plt.ylabel('Count')
plt.legend()
plt.show()

# 기초 통계
print(f"평균 승률: {coc_df_active['win_rate'].mean():.2%}")
print(f"중앙값 승률: {coc_df_active['win_rate'].median():.2%}")
```




```
평균 승률: 56.32%
중앙값 승률: 53.62%
```

In [15]:
```
# 100% 승률 클랜만 필터링
perfect_clans = coc_df_active[coc_df_active['win_rate'] == 1.0]

# 전쟁 횟수 분포 확인
print(f"100% 승률 클랜 수: {len(perfect_clans):,}개")
print(f"\n전쟁 횟수 통계:")
print(perfect_clans['war_total'].describe())

# 시각화
plt.figure(figsize=(10, 6))
sns.histplot(perfect_clans['war_total'], bins=50, color='green')
plt.title('100% 승률 클랜의 전쟁 횟수 분포')
plt.xlabel('총 전쟁 횟수 (war_total)')
plt.ylabel('Count')
plt.show()

# 1~3판만 한 클랜 비율 확인
few_wars = perfect_clans[perfect_clans['war_total'] <= 3]
print(f"\n1~3판만 하고 100%인 클랜: {len(few_wars):,}개 ({len(few_wars)/len(perfe
```

```
100% 승률 클랜 수: 79,735개

전쟁 횟수 통계:
count    79735.000000
mean       176.995648
std        187.994378
min          1.000000
25%         20.000000
50%        114.000000
75%        280.000000
max       1565.000000
Name: war_total, dtype: float64
```




```
1~3판만 하고 100%인 클랜: 12,898개 (16.2%)
```

In [16]:

```
# 20판 이상 클랜만 필터링 (Q1 기반)
MIN_WARS = 20
coc_df_reliable = coc_df_active[coc_df_active['war_total'] >= MIN_WARS].copy()

print(f"기준: {MIN_WARS}판 이상")
print(f"전체: {len(coc_df_active):,}개 → 필터 후: {len(coc_df_reliable):,}개")
print(f"제외: {len(coc_df_active) - len(coc_df_reliable):,}개 ({(len(coc_df_acti
```

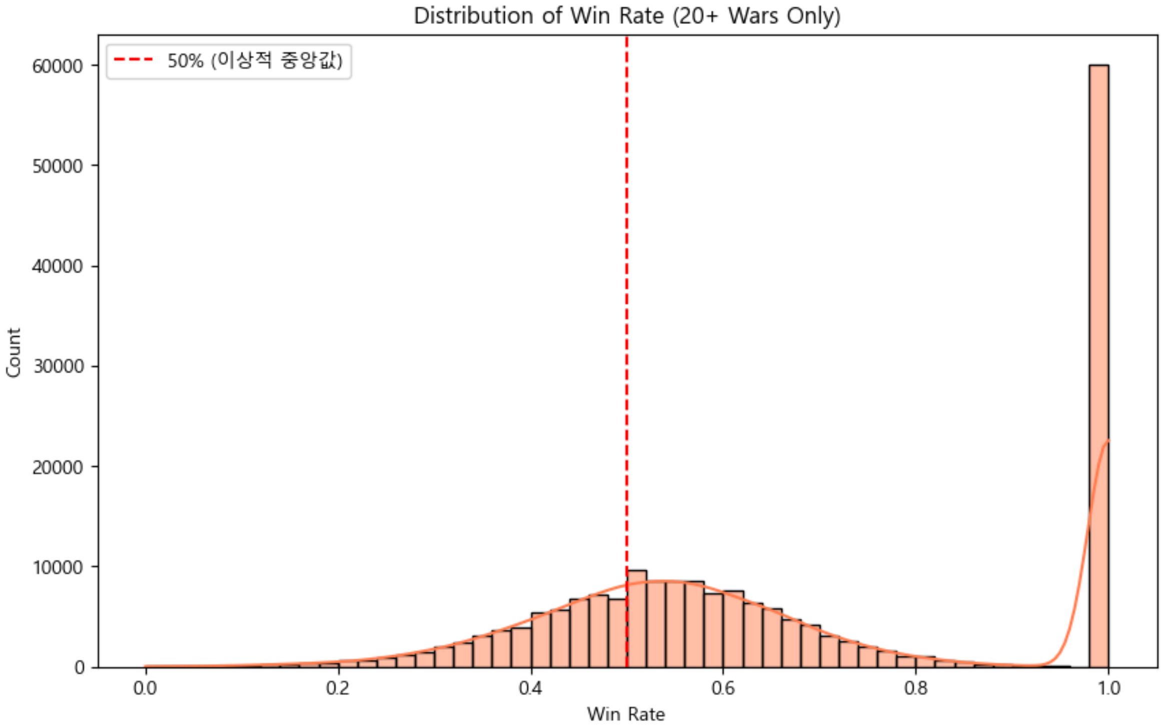
```
기준: 20판 이상
전체: 337,006개 → 필터 후: 197,861개
제외: 139,145개 (41.3%)
```

In [17]:

```
# 20판 이상 클랜만 필터링
MIN_WARS = 20
coc_df_reliable = coc_df_active[coc_df_active['war_total'] >= MIN_WARS].copy()

print(f"기준: {MIN_WARS}판 이상")
print(f"전체: {len(coc_df_active):,}개 → 필터 후: {len(coc_df_reliable):,}개")

# 승률 분포 재시각화
plt.figure(figsize=(10, 6))
sns.histplot(coc_df_reliable['win_rate'], bins=50, kde=True, color='coral')
plt.axvline(x=0.5, color='red', linestyle='--', label='50% (이상적 중앙값)')
plt.title(f'Distribution of Win Rate ({MIN_WARS}+ Wars Only)')
```

```
plt.xlabel('Win Rate')
plt.ylabel('Count')
plt.legend()
plt.show()

# 기초 통계
print(f"\n평균 승률: {coc_df_reliable['win_rate'].mean():.2%}")
print(f"중앙값 승률: {coc_df_reliable['win_rate'].median():.2%}")
```

```
기준: 20판 이상
전체: 337,006개 → 필터 후: 197,861개
```




```
평균 승률: 67.21%
중앙값 승률: 60.61%
```

In [18]:
```
# 20판 이상 + 100% 승률 클랜 추출
perfect_elite = coc_df_reliable[coc_df_reliable['win_rate'] == 1.0]

print(f"20판 이상 100% 승률 클랜: {len(perfect_elite):,}개")
print("=" * 50)

# 주요 지표 확인
print("\n[기본 스펙]")
print(perfect_elite[['clan_level', 'clan_points', 'num_members', 'war_total']].d

print("\n[전쟁 리그 등급 분포]")
print(perfect_elite['clan_war_league'].value_counts())

print("\n[캐피탈 홀 레벨 분포]")
print(perfect_elite['clan_capital_hall_level'].value_counts().sort_index())

# 일반 클랜과 비교
normal_clans = coc_df_reliable[(coc_df_reliable['win_rate'] >= 0.4) & (coc_df_re

print("\n" + "=" * 50)
print("[비교: 100% 승률 vs 일반 클랜(40~60%)]")
print(f"평균 클랜 레벨 - 100%: {perfect_elite['clan_level'].mean():.1f} / 일반: {
print(f"평균 멤버 수 - 100%: {perfect_elite['num_members'].mean():.1f} / 일반: {n
print(f"평균 캐피탈 - 100%: {perfect_elite['clan_capital_hall_level'].mean():.1f}
```

```
20판 이상 100% 승률 클랜: 60,043개
==================================================

[기본 스펙]
         clan_level  clan_points   num_members     war_total
count  60043.000000  60043.00000  60043.000000  60043.000000
mean      15.133105  26717.03211     33.822377    233.528205
std        5.932504  11158.40633     13.055108    184.346770
min        1.000000    512.00000      5.000000     20.000000
25%       10.000000  17992.50000     23.000000     84.000000
50%       15.000000  27266.00000     37.000000    185.000000
75%       20.000000  35761.00000     46.000000    339.000000
max       36.000000  54721.00000     50.000000   1565.000000

[전쟁 리그 등급 분포]
clan_war_league
Crystal League I       9056
Crystal League II      7645
Crystal League III     6912
Gold League I          6521
Gold League II         6011
Master League III      5463
Gold League III        4650
Silver League I        2981
Master League II       2939
Unranked               2241
Silver League II       1687
Master League I        1515
Silver League III       987
Champion League III     644
Champion League II      273
Bronze League I         221
Champion League I       220
Bronze League II         59
Bronze League III        18
Name: count, dtype: int64

[캐피탈 홀 레벨 분포]
clan_capital_hall_level
0         7
2      1475
3      4444
4      3747
5      7469
6      5751
7      9979
8      6075
9      7690
10    13406
Name: count, dtype: int64

==================================================
[비교: 100% 승률 vs 일반 클랜(40~60%)]
평균 클랜 레벨 - 100%: 15.1 / 일반: 10.4
평균 멤버 수 - 100%: 33.8 / 일반: 29.5
평균 캐피탈 - 100%: 7.1 / 일반: 5.6
```

In [19]:

```python
# 전체 대비 비율
reliable_total = len(coc_df_reliable)
perfect_count = len(perfect_elite)
```

```
print(f"20판 이상 전체 클랜: {reliable_total:,}개")
print(f"100% 승률 클랜: {perfect_count:,}개")
print(f"비율: {perfect_count/reliable_total*100:.1f}%")
```

```
20판 이상 전체 클랜: 197,861개
100% 승률 클랜: 60,043개
비율: 30.3%
```

In [20]:

```
# 전쟁 리그 분포 확인
print("전쟁 리그 등급 분포:")
print(coc_df_reliable['clan_war_league'].value_counts())

# 리그별 클랜 특성 비교
league_stats = coc_df_reliable.groupby('clan_war_league').agg({
    'clan_level': 'mean',
    'clan_points': 'mean',
    'num_members': 'mean',
    'clan_capital_hall_level': 'mean'
}).round(1)

print("\n리그별 평균 스펙:")
display(league_stats)
```

```
전쟁 리그 등급 분포:
clan_war_league
Gold League II         23018
Gold League III        22892
Gold League I          20897
Crystal League III     18987
Silver League I        18662
Crystal League I       18550
Crystal League II      17831
Unranked               15508
Silver League II       12425
Master League III       9992
Silver League III       7407
Master League II        5126
Master League I         2508
Bronze League I         1614
Champion League III     1138
Champion League II       481
Champion League I        369
Bronze League II         361
Bronze League III         95
Name: count, dtype: int64

리그별 평균 스펙:
```

clan_war_league	clan_level	clan_points	num_members	clan_capital_hall_level
Bronze League I	5.9	9663.8	21.5	3.3
Bronze League II	5.9	9001.3	20.4	2.9
Bronze League III	5.1	7086.8	19.5	2.5
Champion League I	22.4	30594.6	27.7	9.4
Champion League II	22.5	35559.1	31.2	9.3
Champion League III	22.6	38168.9	34.1	9.5
Crystal League I	18.3	34423.2	39.6	8.7
Crystal League II	16.0	30533.7	37.9	7.9
Crystal League III	13.9	26838.2	36.2	7.1
Gold League I	11.9	23228.6	33.7	6.3
Gold League II	9.9	19755.6	30.8	5.5
Gold League III	8.2	16647.9	27.7	4.8
Master League I	22.9	40132.0	37.7	9.5
Master League II	21.8	39213.4	39.1	9.4
Master League III	20.3	37196.6	39.6	9.1
Silver League I	7.0	14171.0	25.5	4.3
Silver League II	6.3	12343.1	24.1	3.9
Silver League III	6.0	10936.0	22.9	3.6
Unranked	4.6	7518.1	10.9	2.9

In [21]:

```
# 가입 유형별 평균 스펙
type_stats = coc_df_reliable.groupby('clan_type').agg({
    'clan_level': 'mean',
    'clan_points': 'mean',
    'num_members': 'mean',
    'clan_capital_hall_level': 'mean',
    'mean_member_trophies': 'mean'
}).round(1)

print("클랜 가입 유형별 평균 스펙:")
display(type_stats)
```

클랜 가입 유형별 평균 스펙:

	clan_level	clan_points	num_members	clan_capital_hall_level	mean_member_t
clan_type					
closed	13.0	22219.0	28.6	6.2	
inviteOnly	13.0	23798.2	30.1	6.4	
open	10.1	20696.2	31.4	5.7	

In [22]:
```
# 가족 친화 여부별 평균 스펙
family_stats = coc_df_reliable.groupby('isFamilyFriendly').agg({
    'clan_level': 'mean',
    'clan_points': 'mean',
    'num_members': 'mean',
    'clan_capital_hall_level': 'mean',
    'mean_member_trophies': 'mean'
}).round(1)

print("가족 친화 여부별 평균 스펙:")
display(family_stats)
```

가족 친화 여부별 평균 스펙:

	clan_level	clan_points	num_members	clan_capital_hall_level	mean_mem
isFamilyFriendly					
0	12.0	22410.8	30.1	6.1	
1	11.2	21945.2	31.3	6.0	

In [23]:
```
# 예시: 클랜 레벨 구간별로 나눠서 분석
coc_df_reliable['level_group'] = pd.cut(
    coc_df_reliable['clan_level'],
    bins=[0, 5, 10, 15, 20, 40],
    labels=['1-5', '6-10', '11-15', '16-20', '21+']
)

level_stats = coc_df_reliable.groupby('level_group', observed=True).agg({
    'clan_points': 'mean',
    'clan_capital_points': 'mean',
    'mean_member_trophies': 'mean'
}).round(0)

print("클랜 레벨 구간별 스펙:")
display(level_stats)
```

클랜 레벨 구간별 스펙:

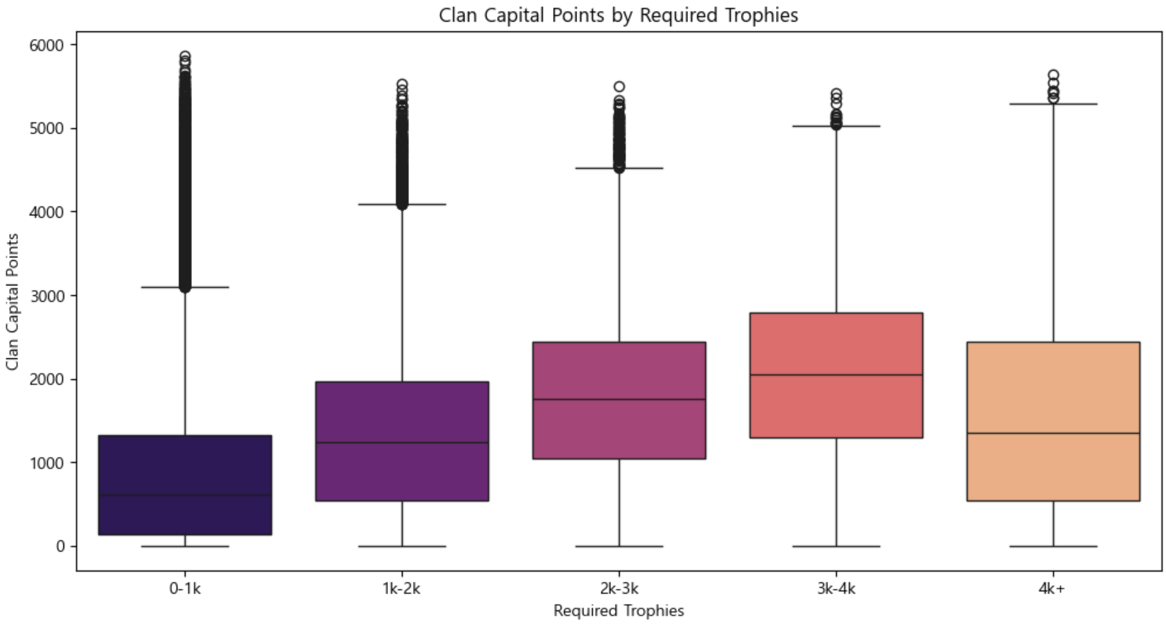
level_group	clan_points	clan_capital_points	mean_member_trophies
1-5	10554.0	347.0	1424.0
6-10	17641.0	788.0	1861.0
11-15	24452.0	1289.0	2343.0
16-20	31048.0	1814.0	2840.0
21+	37400.0	2494.0	3387.0

In [24]:
```
# 1. 요구 트로피 구간화 (0, 1000, 2000, 3000, 4000, 5000+)
bins = [0, 1000, 2000, 3000, 4000, 6000]
labels = ['0-1k', '1k-2k', '2k-3k', '3k-4k', '4k+']
coc_df_reliable['trophy_req_group'] = pd.cut(
    coc_df_reliable['required_trophies'],
    bins=bins,
    labels=labels,
    include_lowest=True
)

# 2. 구간별 평균 성과 (캐피탈 점수)
req_stats = coc_df_reliable.groupby('trophy_req_group', observed=True)['clan_cap

print("요구 트로피 구간별 평균 캐피탈 점수:")
display(req_stats)

# 3. 시각화 (Boxplot)
plt.figure(figsize=(12, 6))
sns.boxplot(data=coc_df_reliable, x='trophy_req_group', y='clan_capital_points',
plt.title('Clan Capital Points by Required Trophies')
plt.xlabel('Required Trophies')
plt.ylabel('Clan Capital Points')
plt.show()
```

요구 트로피 구간별 평균 캐피탈 점수:

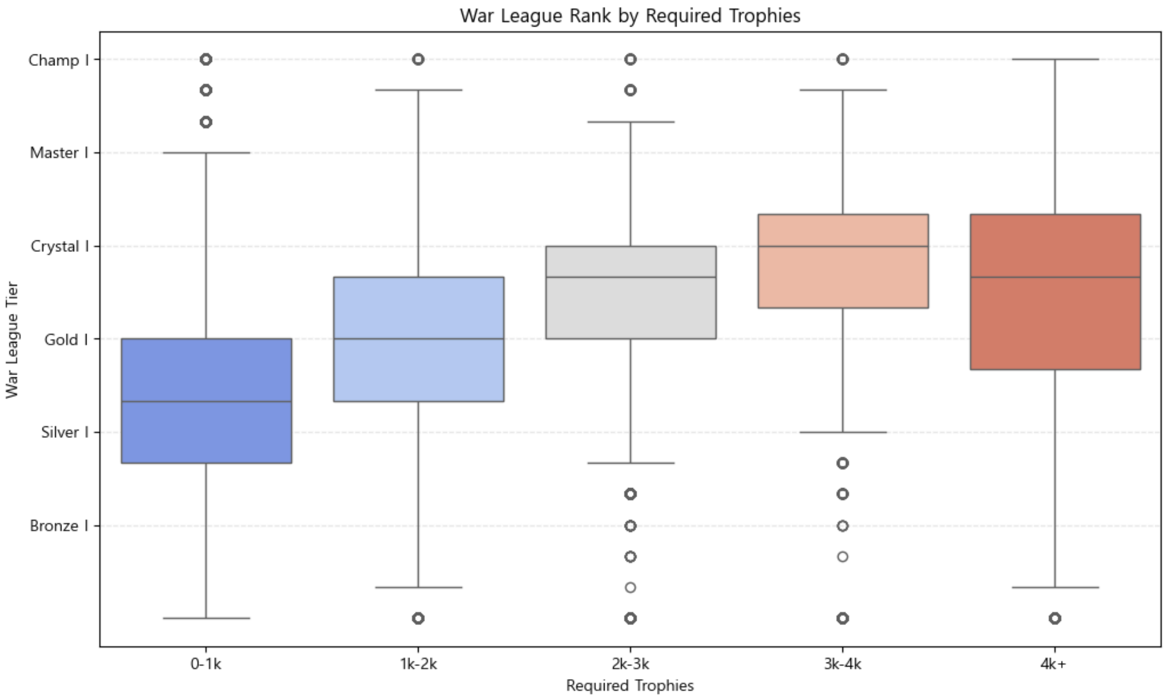
trophy_req_group	mean	median	count
0-1k	883.0	615.0	109551
1k-2k	1337.0	1244.0	49930
2k-3k	1771.0	1762.0	22166
3k-4k	2053.0	2052.0	8209
4k+	1604.0	1345.0	8005

```
C:\Users\tw2ps\AppData\Local\Temp\ipykernel_22452\1217438588.py:19: FutureWarnin
g:

Passing `palette` without assigning `hue` is deprecated and will be removed in v
0.14.0. Assign the `x` variable to `hue` and set `legend=False` for the same effe
ct.

  sns.boxplot(data=coc_df_reliable, x='trophy_req_group', y='clan_capital_point
s', palette='magma')
```



In [25]:
```
# 1. 전쟁 리그 순서 정의 (낮은 등급 -> 높은 등급)
league_order = [
    'Unranked',
    'Bronze League III', 'Bronze League II', 'Bronze League I',
    'Silver League III', 'Silver League II', 'Silver League I',
    'Gold League III', 'Gold League II', 'Gold League I',
    'Crystal League III', 'Crystal League II', 'Crystal League I',
    'Master League III', 'Master League II', 'Master League I',
    'Champion League III', 'Champion League II', 'Champion League I'
]

# 등급을 숫자로 매핑 (Unranked=0, Bronze III=1, ... Champion I=18)
league_map = {name: i for i, name in enumerate(league_order)}
coc_df_reliable['league_rank'] = coc_df_reliable['clan_war_league'].map(league_m

# 2. 시각화 (요구 트로피별 리그 등급 분포)
plt.figure(figsize=(12, 7))
sns.boxplot(data=coc_df_reliable, x='trophy_req_group', y='league_rank', hue='tr

# Y축을 숫자가 아닌 '리그 이름'으로 표시 (가독성 UP)
# 주요 지점(각 티어 1단계)만 표시
ytick_locs = [3, 6, 9, 12, 15, 18]
ytick_labels = ['Bronze I', 'Silver I', 'Gold I', 'Crystal I', 'Master I', 'Cham
plt.yticks(ytick_locs, ytick_labels)

plt.title('War League Rank by Required Trophies')
plt.xlabel('Required Trophies')
plt.ylabel('War League Tier')
plt.grid(axis='y', linestyle='--', alpha=0.3)
plt.show()
```

```
# 3. 구간별 평균 랭킹 확인
print("요구 트로피별 평균 리그 랭킹 (숫자):")
print(coc_df_reliable.groupby('trophy_req_group', observed=True)['league_rank'].
```

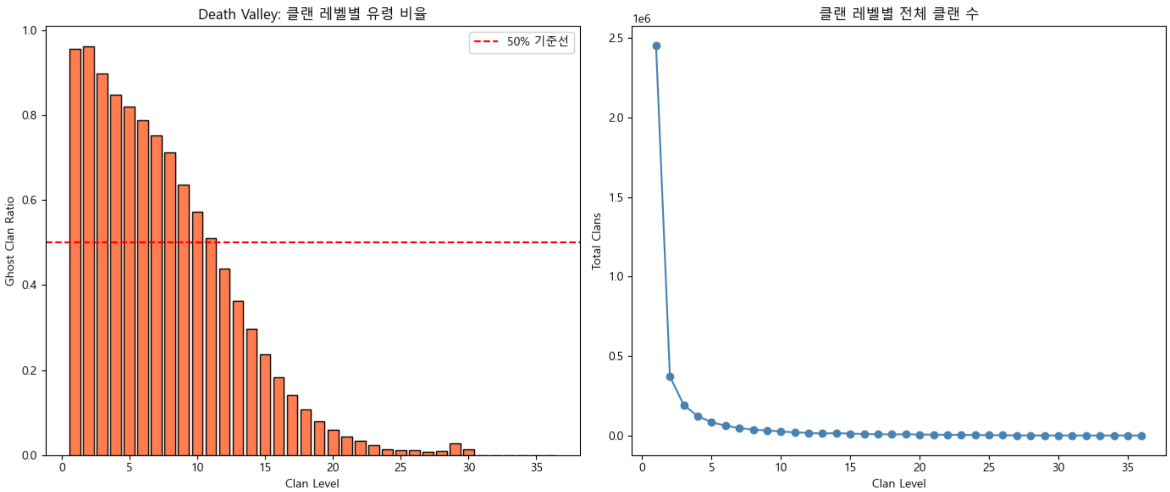



```
요구 트로피별 평균 리그 랭킹 (숫자):
trophy_req_group
0-1k       7.0
1k-2k      8.9
2k-3k     10.5
3k-4k     11.5
4k+       10.4
Name: league_rank, dtype: float64
```

In [26]:

```
# clan_level별 is_ghost 비율 계산
death_valley = coc_df_clean.groupby('clan_level').agg(
    total_clans=('is_ghost', 'count'),
    ghost_clans=('is_ghost', 'sum')
).reset_index()
death_valley['ghost_ratio'] = death_valley['ghost_clans'] / death_valley['total_
# 시각화
plt.figure(figsize=(14, 6))
# 막대 그래프: 유령 클랜 비율
plt.subplot(1, 2, 1)
plt.bar(death_valley['clan_level'], death_valley['ghost_ratio'], color='coral',
plt.axhline(y=0.5, color='red', linestyle='--', label='50% 기준선')
plt.xlabel('Clan Level')
plt.ylabel('Ghost Clan Ratio')
plt.title('Death Valley: 클랜 레벨별 유령 비율')
plt.legend()
plt.xticks(range(0, death_valley['clan_level'].max() + 1, 5))
# 라인 그래프: 전체 클랜 수 분포
plt.subplot(1, 2, 2)
plt.plot(death_valley['clan_level'], death_valley['total_clans'], marker='o', cc
plt.xlabel('Clan Level')
plt.ylabel('Total Clans')
plt.title('클랜 레벨별 전체 클랜 수')
plt.xticks(range(0, death_valley['clan_level'].max() + 1, 5))
plt.tight_layout()
plt.show()
```

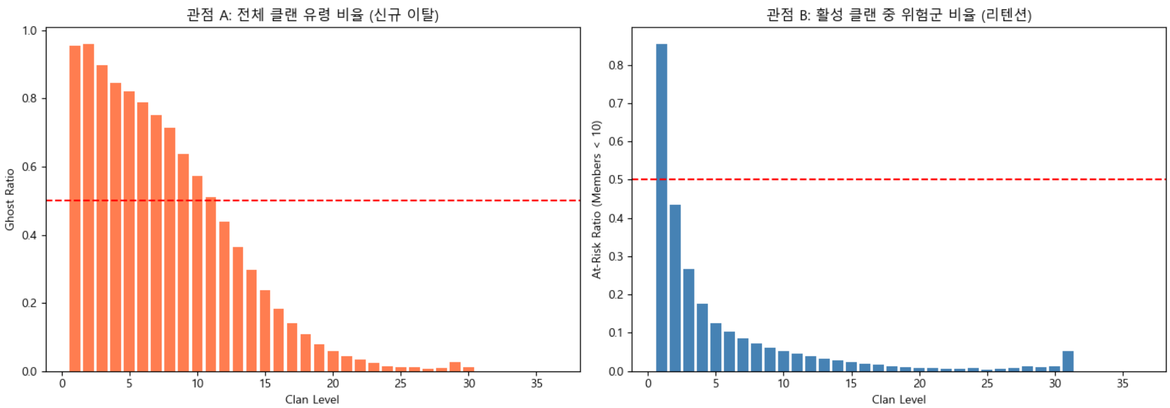
```
# 수치 확인
print("레벨별 유령 비율 (상위 10):")
print(death_valley.sort_values('ghost_ratio', ascending=False).head(10))
```




```
레벨별 유령 비율 (상위 10):
   clan_level  total_clans  ghost_clans  ghost_ratio
1           2       371010       356421     0.960678
0           1      2452736      2342711     0.955142
2           3       191613       171908     0.897163
3           4       122426       103662     0.846732
4           5        83669        68635     0.820316
5           6        62030        48906     0.788425
6           7        47044        35363     0.751701
7           8        36859        26279     0.712960
8           9        32036        20410     0.637096
9          10        24932        14264     0.572116
```

In [27]:
```
# === 관점 A: 전체 클랜 대상 (신규 이탈) ===
death_valley_all = coc_df_clean.groupby('clan_level').agg(
    ghost_ratio_all=('is_ghost', 'mean')
).reset_index()

# === 관점 B: 활성 클랜 대상 (리텐션) ===
# 활성 클랜 중에서 "위험 징후" 정의 (예: 멤버 10명 이하)
coc_df_active['at_risk'] = coc_df_active['num_members'] < 10

death_valley_active = coc_df_active.groupby('clan_level').agg(
    at_risk_ratio=('at_risk', 'mean')
).reset_index()

# === 시각화 ===
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# 관점 A: 전체 클랜 유령 비율
axes[0].bar(death_valley_all['clan_level'], death_valley_all['ghost_ratio_all'],
axes[0].axhline(y=0.5, color='red', linestyle='--')
axes[0].set_xlabel('Clan Level')
axes[0].set_ylabel('Ghost Ratio')
axes[0].set_title('관점 A: 전체 클랜 유령 비율 (신규 이탈)')

# 관점 B: 활성 클랜 위험군 비율
axes[1].bar(death_valley_active['clan_level'], death_valley_active['at_risk_rati
axes[1].axhline(y=0.5, color='red', linestyle='--')
axes[1].set_xlabel('Clan Level')
axes[1].set_ylabel('At-Risk Ratio (Members < 10)')
```

```
axes[1].set_title('관점 B: 활성 클랜 중 위험군 비율 (리텐션)')

plt.tight_layout()
plt.show()

# 수치 비교
print("=== 관점 A: 상위 5 레벨 (유령 비율) ===")
print(death_valley_all.sort_values('ghost_ratio_all', ascending=False).head())

print("\n=== 관점 B: 상위 5 레벨 (위험군 비율) ===")
print(death_valley_active.sort_values('at_risk_ratio', ascending=False).head())
```




```
=== 관점 A: 상위 5 레벨 (유령 비율) ===
   clan_level  ghost_ratio_all
1           2         0.960678
0           1         0.955142
2           3         0.897163
3           4         0.846732
4           5         0.820316

=== 관점 B: 상위 5 레벨 (위험군 비율) ===
   clan_level  at_risk_ratio
0           1       0.855142
1           2       0.435191
2           3       0.266481
3           4       0.175975
4           5       0.124651
```

리텐션의 타겟팅으로 왜 캐피탈 포인트를 근거로 삼았나?

리텐션(생존) 모델링의 핵심은 과거의 누적 성과와 현재의 활동을 분리하는 것이다.

클랜 레벨이나 전쟁 승수는 과거의 데이터도 포함하기 때문에 현재의 활동을 보장하지 못한다는 단점이 있다. 반면 캐피탈 포인트는 매주 진행되는 이벤트의 결과물이므로, **최근 일주일 내 접속 및 활동**을 증명하는 가장 강력한 선행 지표.

설정값(Frequency)이 아닌 **실제 행동** 만을 기반으로 타겟을 정의했기에, 실제 활성 유저를 정확히 타겟팅할 수 있을거라 판단함

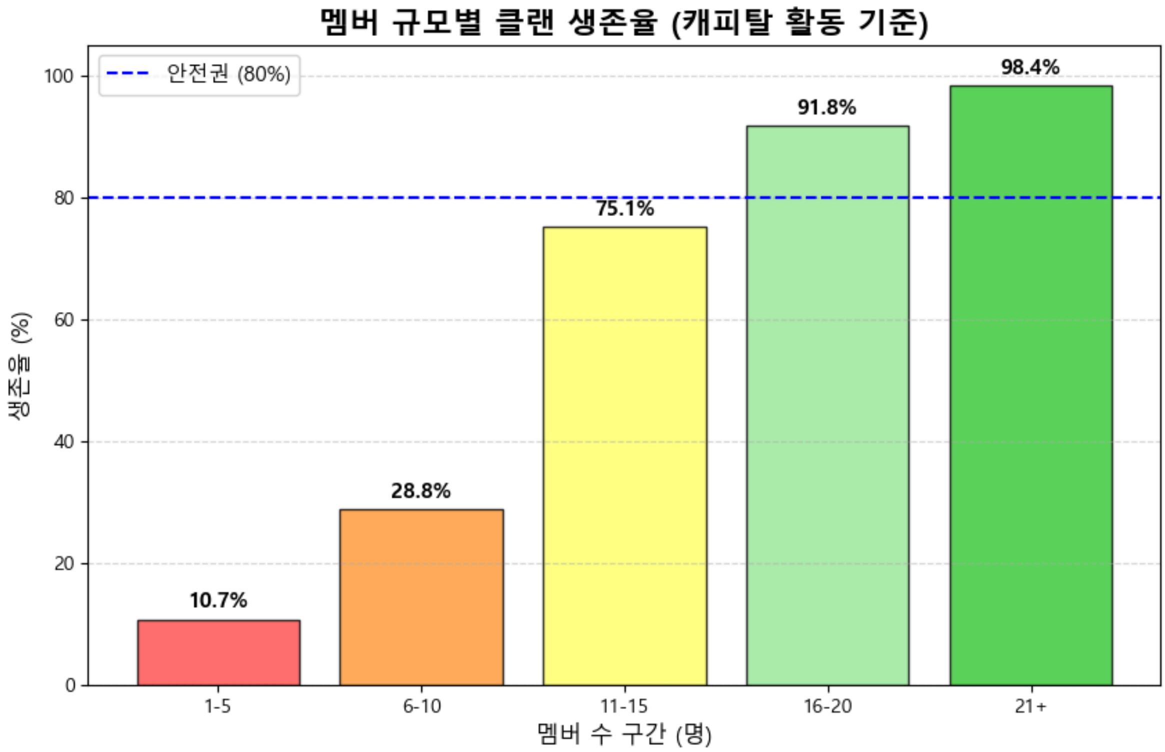
In [28]:
```
# 멤버 수 구간 만들기 (5명 단위)
coc_df_active['member_group'] = pd.cut(coc_df_active['num_members'],
                                       bins=[0, 5, 10, 15, 20, 50],
                                       labels=['1-5', '6-10', '11-15', '16-20',

# 구간별 '캐피탈 포인트 > 0' (실제 생존자) 비율 계산
survival_by_size = coc_df_active.groupby('member_group', observed=False)['clan_c
```

```
print("멤버 규모별 실제 생존율 (Capital Points > 0)")
print(survival_by_size)
```

```
멤버 규모별 실제 생존율 (Capital Points > 0)
member_group
1-5      0.107265
6-10     0.288384
11-15    0.750891
16-20    0.918118
21+      0.984113
Name: clan_capital_points, dtype: float64
```

In [29]:

```
# 그래프 그리기
plt.figure(figsize=(10, 6))

# 데이터 준비 (퍼센트 변환)
survival_pct = survival_by_size * 100

# 막대 그래프 (색상: 위험->안전 그라데이션)
bars = plt.bar(survival_pct.index, survival_pct.values,
               color=['#ff4d4d', '#ff9933', '#ffff66', '#99e699', '#33cc33'],
               edgecolor='black', alpha=0.8)

# 제목 및 라벨 (한글 적용)
plt.title('멤버 규모별 클랜 생존율 (캐피탈 활동 기준)', fontsize=16, fontweight='bo
plt.xlabel('멤버 수 구간 (명)', fontsize=12)
plt.ylabel('생존율 (%)', fontsize=12)
plt.grid(axis='y', linestyle='--', alpha=0.5)
plt.ylim(0, 105)

# 막대 위에 수치 표시
for bar in bars:
    height = bar.get_height()
    plt.text(bar.get_x() + bar.get_width()/2.0, height + 1,
             f'{height:.1f}%', ha='center', va='bottom', fontsize=11, fontweight

# 기준선 (안전권 80%)
plt.axhline(80, color='blue', linestyle='--', label='안전권 (80%)')
plt.legend(fontsize=11)

plt.show()
```



이후 멤버수가 11명은 되어야 생존율이 75%로 안전권에 가깝게 도달된다는 것을 EDA와 시각화를 통해 확인

In [30]:
```
# 1. 정답지(Target) 세팅:
# 캐피탈 포인트가 있는 클랜만 '생존(1)'으로 정의
coc_df_active['is_retained'] = (coc_df_active['clan_capital_points'] > 0).astype

# 데이터 타입 정리 (True/False -> 1/0)
coc_df_active['isFamilyFriendly'] = coc_df_active['isFamilyFriendly'].astype(int

print(f"Target 생성 완료: is_retained")
print(f"   -> 전체 생존율: {coc_df_active['is_retained'].mean():.1%}")
```

```
Target 생성 완료: is_retained
   -> 전체 생존율: 67.4%
```

베이스 모델 설정

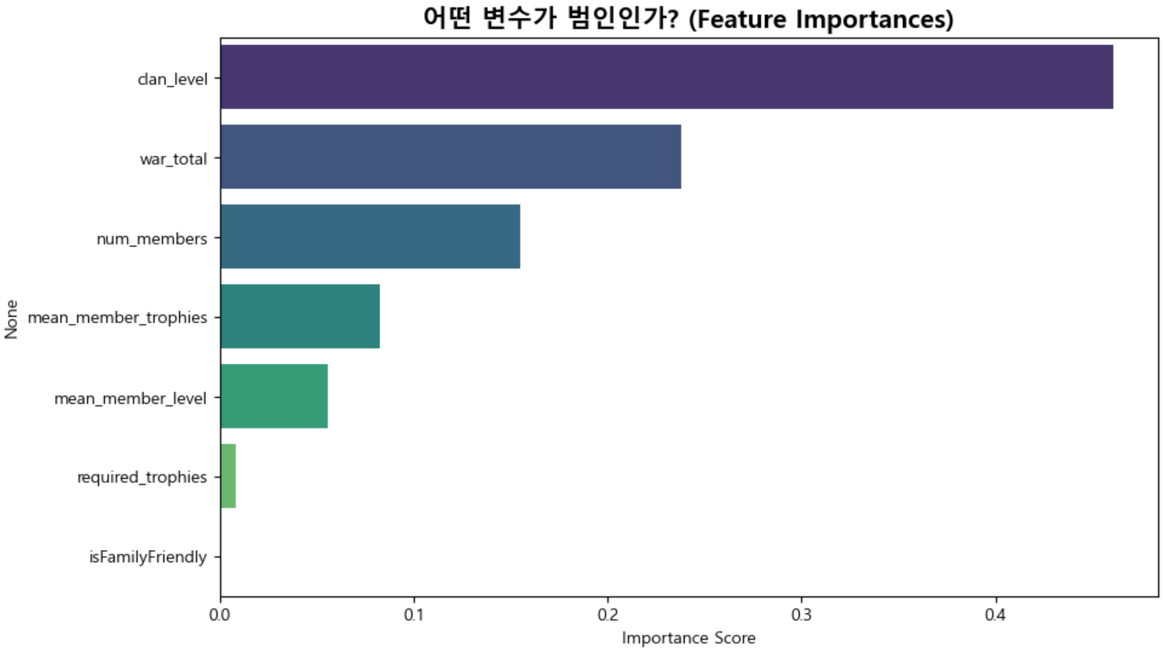
In [31]:
```
# 1. Raw Data Feature 정의 (가공되지 않은 원본)
# 주의: 정답(y)을 유출할 수 있는 'clan_capital_points'나 'war_win_streak' 같은 결고
raw_features = [
    'num_members',          # 멤버 수
    'clan_level',           # 클랜 레벨
    'required_trophies',    # 필요 트로피 (가입 조건)
    'war_total',            # 총 전쟁 횟수 (과거 이력)
    'mean_member_level',    # 멤버 평균 레벨
    'mean_member_trophies', # 멤버 평균 트로피
    'isFamilyFriendly'      # (0/1 변환만 된 상태)
]

# X, y 준비
X_raw = coc_df_active[raw_features]
y_raw = coc_df_active['is_retained']  # 타겟은 아까 정한 '팩트 기반 생존' 그대로

# 데이터 분할
```

```
X_train_raw, X_test_raw, y_train_raw, y_test_raw = train_test_split(
    X_raw, y_raw, test_size=0.2, random_state=42, stratify=y_raw
)

# 베이스라인 모델 학습 우선 Random Forest사용
baseline_model = RandomForestClassifier(n_estimators=100, random_state=42, class
baseline_model.fit(X_train_raw, y_train_raw)

# 성능 평가
y_pred_base = baseline_model.predict(X_test_raw)
print("=== 베이스라인(Raw Data) 성적표 ===")
print(f"AUC Score: {roc_auc_score(y_test_raw, baseline_model.predict_proba(X_tes
print(classification_report(y_test_raw, y_pred_base))
```

```
=== 베이스라인(Raw Data) 성적표 ===
AUC Score: 1.0000
              precision    recall  f1-score   support

           0       1.00      1.00      1.00     22005
           1       1.00      1.00      1.00     45397

    accuracy                           1.00     67402
   macro avg       1.00      1.00      1.00     67402
weighted avg       1.00      1.00      1.00     67402
```

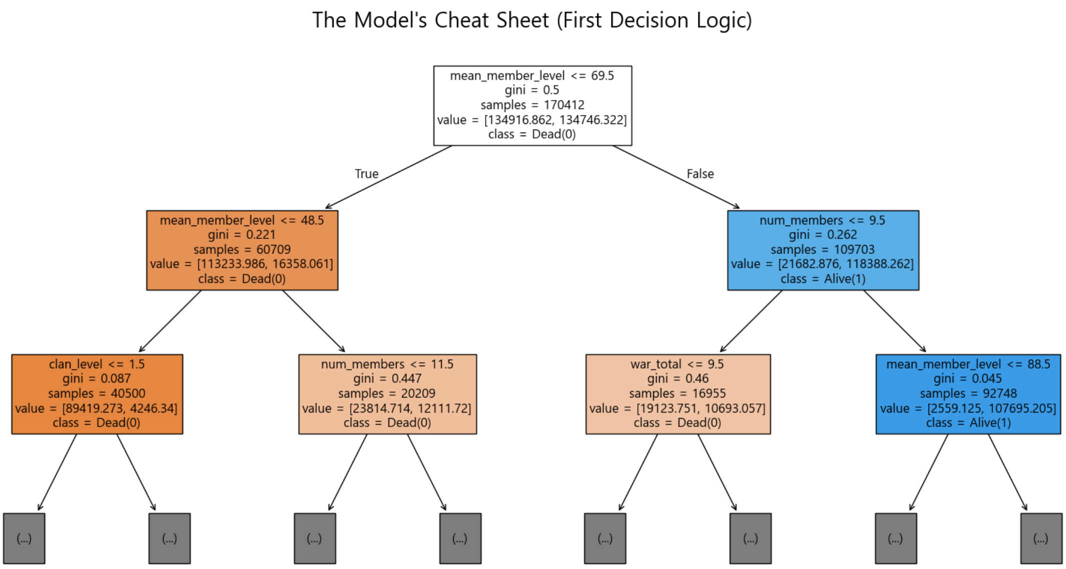
In [32]:

```
# 중요도 추출
importances = pd.Series(baseline_model.feature_importances_, index=raw_features)

# 시각화
plt.figure(figsize=(10, 6))
sns.barplot(
    x=importances.values,
    y=importances.index,
    hue=importances.index,
    palette='viridis',
    legend=False
)
plt.title('어떤 변수가 범인인가? (Feature Importances)', fontsize=15, fontweight=
plt.xlabel('Importance Score')
plt.show()

print("=== 범인 명단 (Top 3) ===")
print(importances.head(3))
```




```
=== 범인 명단 (Top 3) ===
clan_level      0.460868
war_total       0.237973
num_members     0.155065
dtype: float64
```

In [33]:
```
from sklearn.tree import plot_tree

# 랜덤 포레스트 속 첫 번째 나무만 꺼내기
one_tree = baseline_model.estimators_[0]

plt.figure(figsize=(20, 10))
plot_tree(one_tree,
          feature_names=raw_features,
          class_names=['Dead(0)', 'Alive(1)'],
          filled=True,
          max_depth=2, # 너무 깊으면 안 보이니까 위쪽만 봄
          fontsize=12)
plt.title("The Model's Cheat Sheet (First Decision Logic)", fontsize=20)
plt.show()
```



In [34]:
```python
# [수정] 베이스라인 Feature 재정의 (누수 변수 제거)
# clan_level: 직접적인 누수 원인 (제거 필수)
# war_total: 레벨과 상관관계가 높으므로 안전하게 제거
real_raw_features = [
    'num_members',
    'mean_member_level',
    'mean_member_trophies',
    'isFamilyFriendly'
]

# 다시 데이터 준비
X_raw_fix = coc_df_active[real_raw_features]
y_raw_fix = coc_df_active['is_retained']

# 분할 및 학습
X_train_fix, X_test_fix, y_train_fix, y_test_fix = train_test_split(
    X_raw_fix, y_raw_fix, test_size=0.2, random_state=42, stratify=y_raw_fix
)

baseline_model_fix = RandomForestClassifier(n_estimators=100, random_state=42, c
baseline_model_fix.fit(X_train_fix, y_train_fix)

# 결과 확인
print("=== 진짜 베이스라인(Leakage Removed) 성적표 ===")
print(f"AUC Score: {roc_auc_score(y_test_fix, baseline_model_fix.predict_proba(X
print(classification_report(y_test_fix, baseline_model_fix.predict(X_test_fix)))
```

```
=== 진짜 베이스라인(Leakage Removed) 성적표 ===
AUC Score: 0.9752
              precision    recall  f1-score   support

           0       0.89      0.88      0.88     22005
           1       0.94      0.95      0.94     45397

    accuracy                           0.92     67402
   macro avg       0.91      0.91      0.91     67402
weighted avg       0.92      0.92      0.92     67402
```

In [35]:
```python
# [1] activity_ratio (활동 효율성)
# 의미: 레벨 대비 트로피가 얼마나 높은가? (높으면 하드 유저, 낮으면 즐겜 유저)
# 공식: 평균 트로피 / (평균 레벨 + 1)  <-- +1은 0으로 나누기 방지용
coc_df_active['activity_ratio'] = coc_df_active['mean_member_trophies'] / (coc_d

# [2] entry_gap (진입 장벽과의 격차)
# 의미: 클랜원들이 가입 조건보다 얼마나 더 강한가? (높으면 '여기가 좋아서' 남은 찐팬들
# 공식: 평균 트로피 - 가입 조건 트로피
coc_df_active['entry_gap'] = coc_df_active['mean_member_trophies'] - coc_df_acti

# [3] war_frequency_code (전쟁 빈도 숫자 변환)
# 문자로 된 'always', 'never' 등을 숫자로 바꿔줘야 모델이 이해함
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()
coc_df_active['war_frequency_code'] = le.fit_transform(coc_df_active['war_freque

# [4] isFamilyFriendly (숫자로 변환)
# True/False를 1/0으로 변환
coc_df_active['isFamilyFriendly'] = coc_df_active['isFamilyFriendly'].astype(int
```

```
print("파생 변수 생성 완료!")
print(coc_df_active[['activity_ratio', 'entry_gap']].head()) # 잘 만들어졌나 눈으
```

```
파생 변수 생성 완료!
    activity_ratio  entry_gap
11       12.859649       -767
35       18.774194        546
52       17.500000        -25
58       17.052632        572
81       16.734694        420
```

In [36]:
```
# 1. num_members 제거한 Feature 정의
engineered_features_no_num = [
    'activity_ratio',       # 활동 효율성
    'entry_gap',            # 진입 장벽
    'war_frequency_code',   # 전쟁 빈도
    'isFamilyFriendly'      # 문화
]

# 2. X, y 재정의
X_hard = coc_df_active[engineered_features_no_num] # hard mode!
y_hard = coc_df_active['is_retained']

# 3. 데이터 분할
X_train_h, X_test_h, y_train_h, y_test_h = train_test_split(
    X_hard, y_hard, test_size=0.2, random_state=42, stratify=y_hard
)

# 4. 모델 학습
rf_model_hard = RandomForestClassifier(n_estimators=100, random_state=42, class_
rf_model_hard.fit(X_train_h, y_train_h)

# 5. 평가
y_pred_h = rf_model_hard.predict(X_test_h)
y_prob_h = rf_model_hard.predict_proba(X_test_h)[:, 1]


print(f"AUC Score: {roc_auc_score(y_test_h, y_prob_h):.4f}")
print("-" * 30)
print(classification_report(y_test_h, y_pred_h))

# 중요도 확인 (이제 진짜가 보일 것임)
if hasattr(rf_model_hard, 'feature_importances_'):
    importances_h = pd.Series(rf_model_hard.feature_importances_, index=engineer
    print("\n=== 중요도 ===")
    print(importances_h)
```

```
AUC Score: 0.8615
------------------------------
              precision    recall  f1-score   support

           0       0.73      0.68      0.70     22005
           1       0.85      0.88      0.86     45397

    accuracy                           0.81     67402
   macro avg       0.79      0.78      0.78     67402
weighted avg       0.81      0.81      0.81     67402


=== 중요도 ===
activity_ratio        0.479932
entry_gap             0.375311
war_frequency_code    0.137145
isFamilyFriendly      0.007612
dtype: float64
```

In [37]:

```
# 0. 불균형 비율 계산 (XGB/LGBM용)
pos_weight = y_train_h.value_counts()[0] / y_train_h.value_counts()[1]

# =========================================================
# 1. XGBoost
# =========================================================

xgb_model_h = XGBClassifier(
    n_estimators=100,
    max_depth=5,            # 과적합 방지 위해 깊이 제한
    learning_rate=0.1,
    scale_pos_weight=pos_weight, # 불균형 보정
    random_state=42,
    n_jobs=-1,
    eval_metric='logloss'
)
xgb_model_h.fit(X_train_h, y_train_h)

# 평가
xgb_pred_h = xgb_model_h.predict(X_test_h)
xgb_prob_h = xgb_model_h.predict_proba(X_test_h)[:, 1]

print(f"=== XGBoost ===")
print(f"AUC Score: {roc_auc_score(y_test_h, xgb_prob_h):.4f}")
print(classification_report(y_test_h, xgb_pred_h))


# =========================================================
# 2. LightGBM
# =========================================================

lgbm_model_h = LGBMClassifier(
    n_estimators=100,
    max_depth=5,
    learning_rate=0.1,
    scale_pos_weight=pos_weight,
    random_state=42,
    n_jobs=-1,
    verbose=-1
)
lgbm_model_h.fit(X_train_h, y_train_h)
```

```python
# 평가
lgbm_pred_h = lgbm_model_h.predict(X_test_h)
lgbm_prob_h = lgbm_model_h.predict_proba(X_test_h)[:, 1]

print(f"=== LightGBM ===")
print(f"AUC Score: {roc_auc_score(y_test_h, lgbm_prob_h):.4f}")
print(classification_report(y_test_h, lgbm_pred_h))

# ==========================================================
# 3. 중요도 비교 (누가 1등일까?)
# ==========================================================
xgb_imp = pd.Series(xgb_model_h.feature_importances_, index=engineered_features_
lgbm_imp = pd.Series(lgbm_model_h.feature_importances_, index=engineered_feature

# [3] LightGBM (추가: Gain/Power 기준으로 변환!) 👈 여기가 핵심 추가!
lgbm_gain = pd.Series(lgbm_model_h.booster_.feature_importance(importance_type='

print("\n[중요도 1위 비교]")
# .iloc[0]으로 변경하여 "첫 번째 값"을 명확히 가져옵니다.
print(f"XGBoost Pick: {xgb_imp.index[0]} ({xgb_imp.iloc[0]:.3f})")
print(f"LightGBM Pick: {lgbm_imp.index[0]} ({lgbm_imp.iloc[0]:.3f})")
print(f"3. LGBM (Power): {lgbm_gain.index[0]} ({lgbm_gain.iloc[0]:.3f})")
```

```
=== XGBoost ===
AUC Score: 0.8861
              precision    recall  f1-score   support

           0       0.69      0.77      0.73     22005
           1       0.88      0.84      0.86     45397

    accuracy                           0.81     67402
   macro avg       0.79      0.80      0.79     67402
weighted avg       0.82      0.81      0.82     67402

=== LightGBM ===
AUC Score: 0.8862
              precision    recall  f1-score   support

           0       0.69      0.77      0.73     22005
           1       0.88      0.84      0.86     45397

    accuracy                           0.81     67402
   macro avg       0.79      0.80      0.79     67402
weighted avg       0.82      0.81      0.82     67402


[중요도 1위 비교]
XGBoost Pick: war_frequency_code (0.567)
LightGBM Pick: entry_gap (1278.000)
3. LGBM (Power): activity_ratio (230728.081)
```

XGBoost (Gain 기준 - 한 방이 있는 놈) "이 변수로 나눴을 때 정답률이 얼마나 확 올라갔나?"를 봅니다. 의미: war_frequency_code는 한 번 질문하면("전쟁 자주 해?") 정답을 아주 명쾌하게 갈라주는 핵심 질문이라는 뜻입니다.

LightGBM (Split 기준 - 깐깐한 놈) "이 변수를 몇 번이나 사용했나?"를 봅니다. (숫자가 1278로 큰 이유) 의미: entry_gap은 연속된 숫자라서("Gap이 100보다 커? 200보다 커? 300보다 커?") 아주 디테일하게, 자주 물어봐야 하는 변수라는 뜻입니다.

In [38]:
```
# 1. 결과 모으기
model_results = []

# (1) Random Forest (Hard)
model_results.append({
    'Model': 'Random Forest',
    'AUC': roc_auc_score(y_test_h, rf_model_hard.predict_proba(X_test_h)[:, 1]),
    'F1': f1_score(y_test_h, rf_model_hard.predict(X_test_h)),
    'Top Feature': importances_h.index[0] # 아까 구한 RF 중요도
})

# (2) XGBoost (Hard)
model_results.append({
    'Model': 'XGBoost',
    'AUC': roc_auc_score(y_test_h, xgb_model_h.predict_proba(X_test_h)[:, 1]),
    'F1': f1_score(y_test_h, xgb_model_h.predict(X_test_h)),
    'Top Feature': pd.Series(xgb_model_h.feature_importances_, index=engineered_
})

# (3) LightGBM (Hard)
model_results.append({
    'Model': 'LightGBM',
    'AUC': roc_auc_score(y_test_h, lgbm_model_h.predict_proba(X_test_h)[:, 1]),
    'F1': f1_score(y_test_h, lgbm_model_h.predict(X_test_h)),
    'Top Feature': pd.Series(lgbm_model_h.feature_importances_, index=engineered
})

# 2. 데이터프레임으로 변환
comparison_final = pd.DataFrame(model_results).set_index('Model')

display(comparison_final.sort_values(by='AUC', ascending=False))

# 3. 중요도 통합 시각화 (모델별로 뭘 좋아하는지 한눈에 보기)
imp_df = pd.DataFrame({
    'RF': pd.Series(rf_model_hard.feature_importances_, index=engineered_feature
    'XGB': pd.Series(xgb_model_h.feature_importances_, index=engineered_features
    'LGBM': pd.Series(lgbm_model_h.feature_importances_, index=engineered_featur
})

# 정규화 (스케일 맞추기: 0~1 사이로)
imp_df = imp_df / imp_df.max()

print("\n모델별 '가장 중요한' 변수 비교 (1.0 = 가장 중요함)")
display(imp_df.style.background_gradient(cmap='Greens', axis=0))
```

Model	AUC	F1	Top Feature
LightGBM	0.886168	0.857901	entry_gap
XGBoost	0.886075	0.857730	war_frequency_code
Random Forest	0.861484	0.862925	activity_ratio

```
모델별 '가장 중요한' 변수 비교 (1.0 = 가장 중요함)
```

	RF	XGB	LGBM
activity_ratio	1.000000	0.478218	0.884194
entry_gap	0.782007	0.215885	1.000000
war_frequency_code	0.285759	1.000000	0.258998
isFamilyFriendly	0.015860	0.070192	0.144757

하이퍼 파라미티 튜닝

In [ ]:

```
# import optuna
# from sklearn.model_selection import cross_val_score

# # 최적화 목표 함수 정의
# def objective(trial):
#     params = {
#         'n_estimators': trial.suggest_int('n_estimators', 50, 300),
#         'max_depth': trial.suggest_int('max_depth', 3, 12),
#         'learning_rate': trial.suggest_float('learning_rate', 0.01, 0.3, log=T
#         'num_leaves': trial.suggest_int('num_leaves', 20, 150),
#         'min_child_samples': trial.suggest_int('min_child_samples', 5, 100),
#         'subsample': trial.suggest_float('subsample', 0.5, 1.0),
#         'colsample_bytree': trial.suggest_float('colsample_bytree', 0.5, 1.0),
#         'scale_pos_weight': pos_weight,  # 불균형 보정 (고정)
#         'random_state': 42,
#         'n_jobs': -1,
#         'verbose': -1
#     }

#     model = LGBMClassifier(**params)

#     # 3-Fold CV로 AUC 평균 계산
#     score = cross_val_score(model, X_train_h, y_train_h, cv=3, scoring='roc_au
#     return score

# # 튜닝 실행 (50회 시도, 약 5~10분 소요)
# study = optuna.create_study(direction='maximize')
# study.optimize(objective, n_trials=50, show_progress_bar=True)

# # 결과 확인
# print("최적 파라미터:", study.best_params)
# print(f"최고 AUC (CV): {study.best_value:.4f}")
```

```
[I 2026-01-26 19:17:13,817] A new study created in memory with name: no-name-889d
88f5-4789-48e5-abab-c8fb67004e76
  0%|          | 0/50 [00:00<?, ?it/s]
```

```
[I 2026-01-26 19:17:19,373] Trial 0 finished with value: 0.8844552964774185 and parameters: {'n_estimators': 236, 'max_depth': 12, 'learning_rate': 0.02336596590020548, 'num_leaves': 134, 'min_child_samples': 76, 'subsample': 0.8727378214021855, 'colsample_bytree': 0.5734356804501792}. Best is trial 0 with value: 0.8844552964774185.
[I 2026-01-26 19:17:22,249] Trial 1 finished with value: 0.8841377548406409 and parameters: {'n_estimators': 231, 'max_depth': 7, 'learning_rate': 0.1705237089610084, 'num_leaves': 119, 'min_child_samples': 76, 'subsample': 0.737019537572748, 'colsample_bytree': 0.9181787289539609}. Best is trial 0 with value: 0.8844552964774185.
[I 2026-01-26 19:17:23,964] Trial 2 finished with value: 0.8862448545468213 and parameters: {'n_estimators': 107, 'max_depth': 11, 'learning_rate': 0.04484523555970111, 'num_leaves': 35, 'min_child_samples': 86, 'subsample': 0.598342740796656, 'colsample_bytree': 0.9723309901385784}. Best is trial 2 with value: 0.8862448545468213.
[I 2026-01-26 19:17:24,793] Trial 3 finished with value: 0.8804581019678498 and parameters: {'n_estimators': 55, 'max_depth': 4, 'learning_rate': 0.09870276465291472, 'num_leaves': 107, 'min_child_samples': 93, 'subsample': 0.9017039299051479, 'colsample_bytree': 0.5061846741039049}. Best is trial 2 with value: 0.8862448545468213.
[I 2026-01-26 19:17:27,943] Trial 4 finished with value: 0.8863993263507446 and parameters: {'n_estimators': 273, 'max_depth': 6, 'learning_rate': 0.044329453568268465, 'num_leaves': 120, 'min_child_samples': 96, 'subsample': 0.8271204765543994, 'colsample_bytree': 0.7261770660910841}. Best is trial 4 with value: 0.8863993263507446.
[I 2026-01-26 19:17:32,109] Trial 5 finished with value: 0.8854021823923118 and parameters: {'n_estimators': 238, 'max_depth': 9, 'learning_rate': 0.054811775043439925, 'num_leaves': 141, 'min_child_samples': 66, 'subsample': 0.9074761032814787, 'colsample_bytree': 0.8925670441003208}. Best is trial 4 with value: 0.8863993263507446.
[I 2026-01-26 19:17:34,386] Trial 6 finished with value: 0.882252473112772 and parameters: {'n_estimators': 175, 'max_depth': 5, 'learning_rate': 0.014501639874756462, 'num_leaves': 23, 'min_child_samples': 28, 'subsample': 0.7997675862925253, 'colsample_bytree': 0.9234573564220419}. Best is trial 4 with value: 0.8863993263507446.
[I 2026-01-26 19:17:35,151] Trial 7 finished with value: 0.8737167611454106 and parameters: {'n_estimators': 82, 'max_depth': 3, 'learning_rate': 0.0116189214473979, 'num_leaves': 59, 'min_child_samples': 34, 'subsample': 0.9916089880823405, 'colsample_bytree': 0.7220570623803859}. Best is trial 4 with value: 0.8863993263507446.
[I 2026-01-26 19:17:37,018] Trial 8 finished with value: 0.8849002513021368 and parameters: {'n_estimators': 168, 'max_depth': 4, 'learning_rate': 0.031979485919617766, 'num_leaves': 39, 'min_child_samples': 52, 'subsample': 0.692368127994048, 'colsample_bytree': 0.9023830740784586}. Best is trial 4 with value: 0.8863993263507446.
[I 2026-01-26 19:17:40,122] Trial 9 finished with value: 0.8855297750206423 and parameters: {'n_estimators': 241, 'max_depth': 6, 'learning_rate': 0.017571568836119208, 'num_leaves': 110, 'min_child_samples': 87, 'subsample': 0.9898485986518035, 'colsample_bytree': 0.9391987628222023}. Best is trial 4 with value: 0.8863993263507446.
[I 2026-01-26 19:17:44,302] Trial 10 finished with value: 0.8804744335772642 and parameters: {'n_estimators': 300, 'max_depth': 9, 'learning_rate': 0.24622784106422174, 'num_leaves': 82, 'min_child_samples': 49, 'subsample': 0.5243188231213126, 'colsample_bytree': 0.7513960555100977}. Best is trial 4 with value: 0.8863993263507446.
[I 2026-01-26 19:17:46,586] Trial 11 finished with value: 0.8862102143866686 and parameters: {'n_estimators': 120, 'max_depth': 12, 'learning_rate': 0.055766743159166174, 'num_leaves': 76, 'min_child_samples': 98, 'subsample': 0.6021286991970431, 'colsample_bytree': 0.7679223301038567}. Best is trial 4 with value: 0.8863993263507446.
```

```
[I 2026-01-26 19:17:48,745] Trial 12 finished with value: 0.8861741401111408 and parameters: {'n_estimators': 129, 'max_depth': 9, 'learning_rate': 0.03508485918169519, 'num_leaves': 58, 'min_child_samples': 7, 'subsample': 0.6474206842996083, 'colsample_bytree': 0.6434238191069412}. Best is trial 4 with value: 0.8863993263507446.
[I 2026-01-26 19:17:53,389] Trial 13 finished with value: 0.884082264683224 and parameters: {'n_estimators': 293, 'max_depth': 10, 'learning_rate': 0.09892909535277448, 'num_leaves': 95, 'min_child_samples': 80, 'subsample': 0.5174285446624218, 'colsample_bytree': 0.821512840475299}. Best is trial 4 with value: 0.8863993263507446.
[I 2026-01-26 19:17:55,119] Trial 14 finished with value: 0.8865373619968259 and parameters: {'n_estimators': 185, 'max_depth': 7, 'learning_rate': 0.07759070045090116, 'num_leaves': 21, 'min_child_samples': 100, 'subsample': 0.8039814219677399, 'colsample_bytree': 0.9972552385911021}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:17:58,245] Trial 15 finished with value: 0.8858608232068891 and parameters: {'n_estimators': 209, 'max_depth': 7, 'learning_rate': 0.08185099119328569, 'num_leaves': 125, 'min_child_samples': 100, 'subsample': 0.8081505423149917, 'colsample_bytree': 0.6604709944907498}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:18:01,286] Trial 16 finished with value: 0.8847517715121098 and parameters: {'n_estimators': 272, 'max_depth': 6, 'learning_rate': 0.17001274498752783, 'num_leaves': 147, 'min_child_samples': 60, 'subsample': 0.819514138838692, 'colsample_bytree': 0.8324525063131963}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:18:03,640] Trial 17 finished with value: 0.8862765984152005 and parameters: {'n_estimators': 185, 'max_depth': 6, 'learning_rate': 0.07971352364277912, 'num_leaves': 67, 'min_child_samples': 40, 'subsample': 0.7463862434217826, 'colsample_bytree': 0.6755682332974777}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:18:07,391] Trial 18 finished with value: 0.8835293015720541 and parameters: {'n_estimators': 268, 'max_depth': 8, 'learning_rate': 0.14148723011458383, 'num_leaves': 96, 'min_child_samples': 68, 'subsample': 0.8505116144907104, 'colsample_bytree': 0.9993176120681296}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:18:09,637] Trial 19 finished with value: 0.8859352187044998 and parameters: {'n_estimators': 156, 'max_depth': 8, 'learning_rate': 0.026963547718062676, 'num_leaves': 46, 'min_child_samples': 16, 'subsample': 0.9414421503704102, 'colsample_bytree': 0.8313629784585471}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:18:11,950] Trial 20 finished with value: 0.8840709505537064 and parameters: {'n_estimators': 196, 'max_depth': 5, 'learning_rate': 0.042019108964145895, 'num_leaves': 24, 'min_child_samples': 90, 'subsample': 0.7746295390918183, 'colsample_bytree': 0.5959037296211372}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:18:14,544] Trial 21 finished with value: 0.8861882330327623 and parameters: {'n_estimators': 205, 'max_depth': 6, 'learning_rate': 0.08822901722100963, 'num_leaves': 65, 'min_child_samples': 38, 'subsample': 0.7392785225527835, 'colsample_bytree': 0.6945127284382873}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:18:16,697] Trial 22 finished with value: 0.8862139678066221 and parameters: {'n_estimators': 142, 'max_depth': 7, 'learning_rate': 0.07080170900932187, 'num_leaves': 69, 'min_child_samples': 41, 'subsample': 0.7123393843046782, 'colsample_bytree': 0.7838461075873799}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:18:18,531] Trial 23 finished with value: 0.8862043055234965 and parameters: {'n_estimators': 192, 'max_depth': 5, 'learning_rate': 0.13032390075199574, 'num_leaves': 94, 'min_child_samples': 26, 'subsample': 0.8451188291118817, 'colsample_bytree': 0.6639461032570243}. Best is trial 14 with value: 0.8865373619968259.
```

```
[I 2026-01-26 19:18:21,587] Trial 24 finished with value: 0.886273090599588 and p
arameters: {'n_estimators': 259, 'max_depth': 6, 'learning_rate': 0.0597382621950
77274, 'num_leaves': 49, 'min_child_samples': 60, 'subsample': 0.768076921721205
3, 'colsample_bytree': 0.7143043939364162}. Best is trial 14 with value: 0.886537
3619968259.
[I 2026-01-26 19:18:25,047] Trial 25 finished with value: 0.8836238923722849 and
parameters: {'n_estimators': 185, 'max_depth': 8, 'learning_rate': 0.277973511178
52803, 'num_leaves': 107, 'min_child_samples': 46, 'subsample': 0.653031092491535
3, 'colsample_bytree': 0.6087580189404703}. Best is trial 14 with value: 0.886537
3619968259.
[I 2026-01-26 19:18:26,523] Trial 26 finished with value: 0.8844579125343571 and
parameters: {'n_estimators': 154, 'max_depth': 4, 'learning_rate': 0.043880703730
8139, 'num_leaves': 87, 'min_child_samples': 100, 'subsample': 0.680937312943099
7, 'colsample_bytree': 0.7980391071218496}. Best is trial 14 with value: 0.886537
3619968259.
[I 2026-01-26 19:18:30,092] Trial 27 finished with value: 0.8856776596166563 and
parameters: {'n_estimators': 220, 'max_depth': 7, 'learning_rate': 0.069692570987
52457, 'num_leaves': 124, 'min_child_samples': 81, 'subsample': 0.778265975529696
6, 'colsample_bytree': 0.5431554261384876}. Best is trial 14 with value: 0.886537
3619968259.
[I 2026-01-26 19:18:31,323] Trial 28 finished with value: 0.8862432068298105 and
parameters: {'n_estimators': 102, 'max_depth': 5, 'learning_rate': 0.110336613673
67444, 'num_leaves': 36, 'min_child_samples': 69, 'subsample': 0.847487221232962
6, 'colsample_bytree': 0.8523569827370906}. Best is trial 14 with value: 0.886537
3619968259.
[I 2026-01-26 19:18:33,049] Trial 29 finished with value: 0.882908464476479 and p
arameters: {'n_estimators': 250, 'max_depth': 3, 'learning_rate': 0.0236534572610
67157, 'num_leaves': 137, 'min_child_samples': 75, 'subsample': 0.900132980408202
9, 'colsample_bytree': 0.6914034524587077}. Best is trial 14 with value: 0.886537
3619968259.
[I 2026-01-26 19:18:35,692] Trial 30 finished with value: 0.8852626368942476 and
parameters: {'n_estimators': 211, 'max_depth': 6, 'learning_rate': 0.019443553325
10655, 'num_leaves': 75, 'min_child_samples': 93, 'subsample': 0.939613109263822
2, 'colsample_bytree': 0.7342548224059119}. Best is trial 14 with value: 0.886537
3619968259.
[I 2026-01-26 19:18:38,597] Trial 31 finished with value: 0.8861578970018771 and
parameters: {'n_estimators': 261, 'max_depth': 6, 'learning_rate': 0.071442482016
198, 'num_leaves': 52, 'min_child_samples': 57, 'subsample': 0.7769770799079736,
'colsample_bytree': 0.710849250053985}. Best is trial 14 with value: 0.8865373619
968259.
[I 2026-01-26 19:18:41,626] Trial 32 finished with value: 0.8857617968879841 and
parameters: {'n_estimators': 283, 'max_depth': 7, 'learning_rate': 0.059271925569
07739, 'num_leaves': 20, 'min_child_samples': 23, 'subsample': 0.753103248655404
1, 'colsample_bytree': 0.624324194430228}. Best is trial 14 with value: 0.8865373
619968259.
[I 2026-01-26 19:18:44,691] Trial 33 finished with value: 0.8863113448130736 and
parameters: {'n_estimators': 255, 'max_depth': 6, 'learning_rate': 0.037181382666
70898, 'num_leaves': 48, 'min_child_samples': 57, 'subsample': 0.742707322269777
8, 'colsample_bytree': 0.6762761847907643}. Best is trial 14 with value: 0.886537
3619968259.
[I 2026-01-26 19:18:47,640] Trial 34 finished with value: 0.8853272918381863 and
parameters: {'n_estimators': 231, 'max_depth': 8, 'learning_rate': 0.035787180589
33734, 'num_leaves': 29, 'min_child_samples': 43, 'subsample': 0.713214901830888
6, 'colsample_bytree': 0.5794114335889191}. Best is trial 14 with value: 0.886537
3619968259.
[I 2026-01-26 19:18:49,836] Trial 35 finished with value: 0.8862088335778529 and
parameters: {'n_estimators': 223, 'max_depth': 5, 'learning_rate': 0.046389469664
09843, 'num_leaves': 42, 'min_child_samples': 82, 'subsample': 0.815174501780119
5, 'colsample_bytree': 0.6714324644568529}. Best is trial 14 with value: 0.886537
3619968259.
```

```
[I 2026-01-26 19:18:52,967] Trial 36 finished with value: 0.8856491697007859 and parameters: {'n_estimators': 282, 'max_depth': 7, 'learning_rate': 0.126810133539694, 'num_leaves': 32, 'min_child_samples': 33, 'subsample': 0.7236614496867391, 'colsample_bytree': 0.5534279998105497}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:18:55,030] Trial 37 finished with value: 0.885030652660586 and parameters: {'n_estimators': 246, 'max_depth': 4, 'learning_rate': 0.029314286635393136, 'num_leaves': 57, 'min_child_samples': 74, 'subsample': 0.8844637859466586, 'colsample_bytree': 0.6331201953311678}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:18:57,278] Trial 38 finished with value: 0.8862686024313017 and parameters: {'n_estimators': 180, 'max_depth': 6, 'learning_rate': 0.049969263391251476, 'num_leaves': 68, 'min_child_samples': 95, 'subsample': 0.6726259158284729, 'colsample_bytree': 0.7558152172846387}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:18:59,535] Trial 39 finished with value: 0.8863700922077343 and parameters: {'n_estimators': 165, 'max_depth': 10, 'learning_rate': 0.035963712302274284, 'num_leaves': 44, 'min_child_samples': 90, 'subsample': 0.8321524975521466, 'colsample_bytree': 0.8587747756323155}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:19:01,686] Trial 40 finished with value: 0.8858960181118872 and parameters: {'n_estimators': 164, 'max_depth': 10, 'learning_rate': 0.022413916748837744, 'num_leaves': 39, 'min_child_samples': 87, 'subsample': 0.8400238896081318, 'colsample_bytree': 0.8751600533642674}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:19:03,318] Trial 41 finished with value: 0.8863004166868631 and parameters: {'n_estimators': 136, 'max_depth': 11, 'learning_rate': 0.039408854820233784, 'num_leaves': 29, 'min_child_samples': 91, 'subsample': 0.7927735451859044, 'colsample_bytree': 0.9588710704575782}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:19:04,508] Trial 42 finished with value: 0.8849049328005499 and parameters: {'n_estimators': 87, 'max_depth': 11, 'learning_rate': 0.0344115515295967, 'num_leaves': 27, 'min_child_samples': 91, 'subsample': 0.8015443469930679, 'colsample_bytree': 0.9522366275178494}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:19:06,556] Trial 43 finished with value: 0.8863347479880602 and parameters: {'n_estimators': 139, 'max_depth': 11, 'learning_rate': 0.039454882279121584, 'num_leaves': 29, 'min_child_samples': 95, 'subsample': 0.8688542856308537, 'colsample_bytree': 0.9826046283652663}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:19:07,926] Trial 44 finished with value: 0.8843032107051223 and parameters: {'n_estimators': 53, 'max_depth': 10, 'learning_rate': 0.02844785148511548, 'num_leaves': 44, 'min_child_samples': 85, 'subsample': 0.8724144571143003, 'colsample_bytree': 0.9975917449079078}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:19:09,962] Trial 45 finished with value: 0.8863891850593816 and parameters: {'n_estimators': 114, 'max_depth': 12, 'learning_rate': 0.05001752853778461, 'num_leaves': 34, 'min_child_samples': 96, 'subsample': 0.9354025043912646, 'colsample_bytree': 0.9225930438339606}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:19:11,905] Trial 46 finished with value: 0.8863529638154174 and parameters: {'n_estimators': 112, 'max_depth': 12, 'learning_rate': 0.04813322723505472, 'num_leaves': 34, 'min_child_samples': 96, 'subsample': 0.9409182207705907, 'colsample_bytree': 0.9183039925806524}. Best is trial 14 with value: 0.8865373619968259.
[I 2026-01-26 19:19:13,853] Trial 47 finished with value: 0.8862841573416226 and parameters: {'n_estimators': 100, 'max_depth': 12, 'learning_rate': 0.04896292445792345, 'num_leaves': 34, 'min_child_samples': 97, 'subsample': 0.9579418673414617, 'colsample_bytree': 0.9166877589460223}. Best is trial 14 with value: 0.8865373619968259.
```

```
[I 2026-01-26 19:19:16,310] Trial 48 finished with value: 0.8861229400994043 and
parameters: {'n_estimators': 73, 'max_depth': 12, 'learning_rate': 0.053787176665
072094, 'num_leaves': 113, 'min_child_samples': 87, 'subsample': 0.97150375445457
84, 'colsample_bytree': 0.8924172861050849}. Best is trial 14 with value: 0.88653
73619968259.
[I 2026-01-26 19:19:17,766] Trial 49 finished with value: 0.8855345038192423 and
parameters: {'n_estimators': 69, 'max_depth': 11, 'learning_rate': 0.064218612763
95889, 'num_leaves': 20, 'min_child_samples': 78, 'subsample': 0.905267542549376
7, 'colsample_bytree': 0.9249638473195173}. Best is trial 14 with value: 0.886537
3619968259.
최적 파라미터: {'n_estimators': 185, 'max_depth': 7, 'learning_rate': 0.0775907004
5090116, 'num_leaves': 21, 'min_child_samples': 100, 'subsample': 0.8039814219677
399, 'colsample_bytree': 0.9972552385911021}
최고 AUC (CV): 0.8865
```

In [48]:

```python
# 1. 최적 파라미터로 최종 모델 생성
best_params = study.best_params
best_params['scale_pos_weight'] = pos_weight  # 불균형 보정 추가
best_params['random_state'] = 42
best_params['n_jobs'] = -1
best_params['verbose'] = -1

lgbm_tuned = LGBMClassifier(**best_params)

# 2. 전체 학습 데이터로 학습
lgbm_tuned.fit(X_train_h, y_train_h)

# 3. 테스트 데이터로 최종 평가
y_pred_tuned = lgbm_tuned.predict(X_test_h)
y_prob_tuned = lgbm_tuned.predict_proba(X_test_h)[:, 1]

auc_tuned = roc_auc_score(y_test_h, y_prob_tuned)
auc_before = 0.8862  # 튜닝 전 LightGBM 점수

print("=== 최종 모델 성적표 (Tuned LightGBM) ===")
print(f"AUC Score: {auc_tuned:.4f}")
print(f"개선폭: {auc_tuned - auc_before:+.4f}")
print("-" * 40)
print(classification_report(y_test_h, y_pred_tuned))

# 4. Feature Importance 확인 (튜닝 후에도 동일한지?)
tuned_imp = pd.Series(lgbm_tuned.feature_importances_, index=engineered_features
print("\n튜닝 후에도 Top Feature는?")
print(tuned_imp)

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_sc

print("\n=== 최종 모델 성능 지표 ===")
print(f"정확도 (Accuracy): {accuracy_score(y_test_h, y_pred_tuned):.4f}")
print(f"정밀도 (Precision): {precision_score(y_test_h, y_pred_tuned):.4f}")
print(f"재현율 (Recall): {recall_score(y_test_h, y_pred_tuned):.4f}")
print(f"F1-Score: {f1_score(y_test_h, y_pred_tuned):.4f}")
print(f"AUC: {auc_tuned:.4f}")
```

```
=== 최종 모델 성적표 (Tuned LightGBM) ===
AUC Score: 0.8862
개선폭: +0.0000
----------------------------------------
              precision    recall  f1-score   support

           0       0.69      0.77      0.73     22005
           1       0.88      0.83      0.86     45397

    accuracy                           0.81     67402
   macro avg       0.79      0.80      0.79     67402
weighted avg       0.82      0.81      0.82     67402


튜닝 후에도 Top Feature는?
entry_gap              1658
activity_ratio         1551
war_frequency_code      354
isFamilyFriendly        137
dtype: int32

=== 최종 모델 성능 지표 ===
정확도 (Accuracy): 0.8129
정밀도 (Precision): 0.8810
재현율 (Recall): 0.8350
F1-Score: 0.8574
AUC: 0.8862
```

In [41]:

```python
# 1. clan_type 값 확인
print("clan_type 종류:", coc_df_active['clan_type'].unique())
print(coc_df_active['clan_type'].value_counts())

# 2. 숫자로 인코딩 (LabelEncoder 사용)
le_clan_type = LabelEncoder()
coc_df_active['clan_type_code'] = le_clan_type.fit_transform(coc_df_active['clan

# 확인
print("\n인코딩 결과:")
print(pd.DataFrame({'원본': le_clan_type.classes_, '코드': range(len(le_clan_type
```

```
clan_type 종류: ['inviteOnly' 'closed' 'open']
clan_type
open          193964
inviteOnly    120149
closed         22893
Name: count, dtype: int64

인코딩 결과:
           원본  코드
0      closed   0
1  inviteOnly   1
2        open   2
```

In [52]:

```python
# 3. Feature 리스트 업데이트
engineered_features_v2 = [
    'activity_ratio',
    'entry_gap',
    'war_frequency_code',
    'isFamilyFriendly',
    'clan_type_code'  # <-- 새로 추가!
```

```
]

# 4. X, y 재정의
X_v2 = coc_df_active[engineered_features_v2]
y_v2 = coc_df_active['is_retained']

# 5. 데이터 분할
X_train_v2, X_test_v2, y_train_v2, y_test_v2 = train_test_split(
    X_v2, y_v2, test_size=0.2, random_state=42, stratify=y_v2
)

# 6. 모델 학습 (이전 최적 파라미터 재활용)
lgbm_v2 = LGBMClassifier(**best_params)
lgbm_v2.fit(X_train_v2, y_train_v2)

# 7. 평가
y_prob_v2 = lgbm_v2.predict_proba(X_test_v2)[:, 1]
auc_v2 = roc_auc_score(y_test_v2, y_prob_v2)

print(f"=== 🆕 clan_type 추가 후 성능 ===")
print(f"AUC Score: {auc_v2:.4f}")
print(f"개선폭: {auc_v2 - 0.8861:+.4f}")

# 8. 중요도 확인
imp_v2 = pd.Series(lgbm_v2.feature_importances_, index=engineered_features_v2).s
print("\nFeature Importance:")
print(imp_v2)

# 예측값 생성
y_pred_v2 = lgbm_v2.predict(X_test_v2)
print("\n=== 📊 clan_type 추가 후 성능 지표 ===")
print(f"정확도 (Accuracy): {accuracy_score(y_test_v2, y_pred_v2):.4f}")
print(f"정밀도 (Precision): {precision_score(y_test_v2, y_pred_v2):.4f}")
print(f"재현율 (Recall): {recall_score(y_test_v2, y_pred_v2):.4f}")
print(f"F1-Score: {f1_score(y_test_v2, y_pred_v2):.4f}")
print(f"AUC: {auc_v2:.4f}")

# 이전 모델(v1) 지표 저장 (위에서 이미 계산됨)
acc_v1 = accuracy_score(y_test_h, y_pred_tuned)
prec_v1 = precision_score(y_test_h, y_pred_tuned)
rec_v1 = recall_score(y_test_h, y_pred_tuned)
f1_v1 = f1_score(y_test_h, y_pred_tuned)
auc_v1 = auc_tuned  # 0.8862

# v2 모델 지표
y_pred_v2 = lgbm_v2.predict(X_test_v2)
acc_v2 = accuracy_score(y_test_v2, y_pred_v2)
prec_v2 = precision_score(y_test_v2, y_pred_v2)
rec_v2 = recall_score(y_test_v2, y_pred_v2)
f1_v2 = f1_score(y_test_v2, y_pred_v2)

# 비교 출력
print("\n=== 📊 clan_type 추가 전후 비교 ===")
print(f"{'지표':<15} {'Before':>10} {'After':>10} {'개선폭':>10}")
print("-" * 47)
print(f"{'정확도':<15} {acc_v1:>10.4f} {acc_v2:>10.4f} {acc_v2-acc_v1:>+10.4f}")
print(f"{'정밀도':<15} {prec_v1:>10.4f} {prec_v2:>10.4f} {prec_v2-prec_v1:>+10.4
print(f"{'재현율':<15} {rec_v1:>10.4f} {rec_v2:>10.4f} {rec_v2-rec_v1:>+10.4f}")
print(f"{'F1-Score':<15} {f1_v1:>10.4f} {f1_v2:>10.4f} {f1_v2-f1_v1:>+10.4f}")
print(f"{'AUC':<15} {auc_v1:>10.4f} {auc_v2:>10.4f} {auc_v2-auc_v1:>+10.4f}")
```

```
=== 🆕 clan_type 추가 후 성능 ===
AUC Score: 0.8912
개선폭: +0.0051

Feature Importance:
entry_gap              1536
activity_ratio         1415
war_frequency_code      330
clan_type_code          291
isFamilyFriendly        128
dtype: int32

=== 📊 clan_type 추가 후 성능 지표 ===
정확도 (Accuracy): 0.8153
정밀도 (Precision): 0.8863
재현율 (Recall): 0.8325
F1-Score: 0.8586
AUC: 0.8912

=== 📊 clan_type 추가 전후 비교 ===
지표                     Before      After        개선폭
-----------------------------------------------
정확도                     0.8129     0.8153    +0.0024
정밀도                     0.8810     0.8863    +0.0053
재현율                     0.8350     0.8325    -0.0025
F1-Score               0.8574     0.8586    +0.0012
AUC                    0.8862     0.8912    +0.0049
```

In [56]:

```python
# 1. 새 파생변수 생성
coc_df_active['capital_per_member'] = coc_df_active['clan_capital_points'] / (cc

# 2. Feature 리스트 (v3)
engineered_features_v3 = [
    'activity_ratio',
    'entry_gap',
    'war_frequency_code',
    'isFamilyFriendly',
    'clan_type_code',
    # 새로 추가
    'required_townhall_level',       # 가입 문턱
]

# 3. X, y 정의
X_v3 = coc_df_active[engineered_features_v3]
y_v3 = coc_df_active['is_retained']

# 4. 데이터 분할
X_train_v3, X_test_v3, y_train_v3, y_test_v3 = train_test_split(
    X_v3, y_v3, test_size=0.2, random_state=42, stratify=y_v3
)

# 5. 모델 학습 (기존 best_params 재활용)
lgbm_v3 = LGBMClassifier(**best_params)
lgbm_v3.fit(X_train_v3, y_train_v3)

# 6. 평가
y_pred_v3 = lgbm_v3.predict(X_test_v3)
y_prob_v3 = lgbm_v3.predict_proba(X_test_v3)[:, 1]
auc_v3 = roc_auc_score(y_test_v3, y_prob_v3)
```

```
print(f"=== 🚀 v3 모델 성능 (Feature 8개) ===")
print(f"AUC Score: {auc_v3:.4f}")
print(f"개선폭 (vs v2): {auc_v3 - auc_v2:+.4f}")

# 7. 전후 비교
acc_v3 = accuracy_score(y_test_v3, y_pred_v3)
prec_v3 = precision_score(y_test_v3, y_pred_v3)
rec_v3 = recall_score(y_test_v3, y_pred_v3)
f1_v3 = f1_score(y_test_v3, y_pred_v3)

print(f"\n=== 📊 v2 vs v3 비교 ===")
print(f"{'지표':<15} {'v2 (5개)':>10} {'v3 (8개)':>10} {'개선폭':>10}")
print("-" * 47)
print(f"{'정확도':<15} {acc_v2:>10.4f} {acc_v3:>10.4f} {acc_v3-acc_v2:>+10.4f}")
print(f"{'정밀도':<15} {prec_v2:>10.4f} {prec_v3:>10.4f} {prec_v3-prec_v2:>+10.4
print(f"{'재현율':<15} {rec_v2:>10.4f} {rec_v3:>10.4f} {rec_v3-rec_v2:>+10.4f}")
print(f"{'F1-Score':<15} {f1_v2:>10.4f} {f1_v3:>10.4f} {f1_v3-f1_v2:>+10.4f}")
print(f"{'AUC':<15} {auc_v2:>10.4f} {auc_v3:>10.4f} {auc_v3-auc_v2:>+10.4f}")

# 8. Feature Importance
imp_v3 = pd.Series(lgbm_v3.feature_importances_, index=engineered_features_v3).s
print("\n📊 Feature Importance (v3):")
print(imp_v3)
```

```
=== 🚀 v3 모델 성능 (Feature 8개) ===
AUC Score: 0.9524
개선폭 (vs v2): +0.0613

=== 📊 v2 vs v3 비교 ===
지표                v2 (5개)   v3 (8개)       개선폭
-----------------------------------------------
정확도                 0.8153     0.8760    +0.0608
정밀도                 0.8863     0.9437    +0.0573
재현율                 0.8325     0.8678    +0.0353
F1-Score            0.8586     0.9041    +0.0455
AUC                 0.8912     0.9524    +0.0613

📊 Feature Importance (v3):
entry_gap                 1220
activity_ratio            1151
required_townhall_level    606
war_frequency_code         354
clan_type_code             264
isFamilyFriendly           105
dtype: int32
```

In [57]:
```
# 1. 타겟 그룹: 멤버 5~15명인 소규모 클랜
# (보통 죽음의 계곡에 있어서 다들 무시하는 구간!)
target_small_clans = coc_df_active[
    (coc_df_active['num_members'] >= 5) &
    (coc_df_active['num_members'] <= 15)
].copy()

print(f"후보군(5~15명): {len(target_small_clans):,}개")

# 2. X_small 만들기 (학습 때랑 똑같은 변수만!)
X_small = target_small_clans[engineered_features_v2]

# 3. 모델로 예측 (성공 확률 계산)
target_small_clans['success_prob'] = lgbm_v2.predict_proba(X_small)[:, 1]
```

```
# 4. Hidden Gems 선정 (생존 확률 85% 이상)
hidden_gems = target_small_clans[target_small_clans['success_prob'] >= 0.85].sor

print(f"발굴된 Hidden Gems: {len(hidden_gems):,}개 (비율: {len(hidden_gems)/len(t

# 5. 상위 10개 클랜 구경하기
cols_to_view = ['clan_tag', 'num_members', 'clan_level', 'activity_ratio', 'entr
display(hidden_gems[cols_to_view].head(10))
```

후보군(5~15명): 155,259개
발굴된 Hidden Gems: 17,279개 (비율: 11.1%)

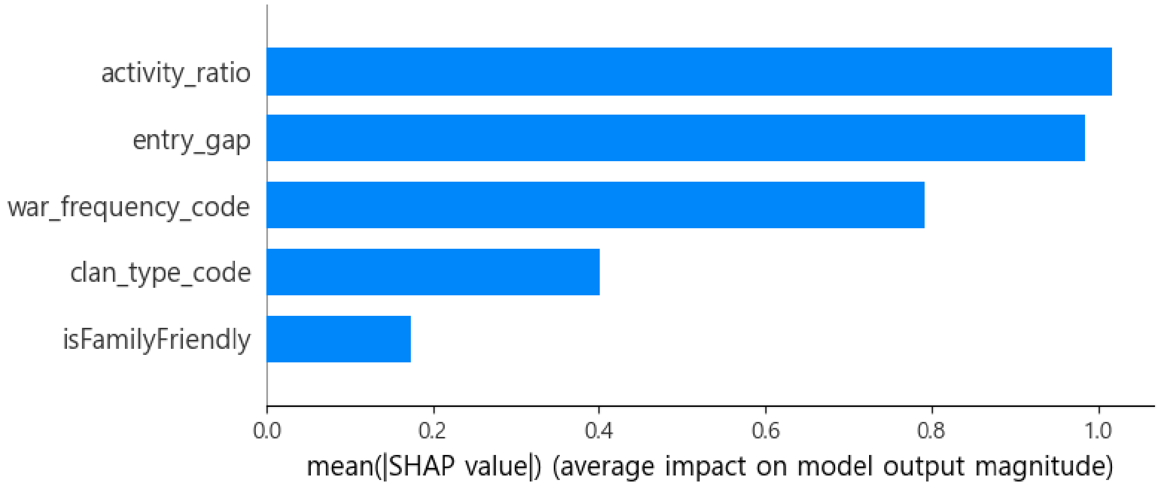
	clan_tag	num_members	clan_level	activity_ratio	entry_gap	clan_type	s
1528928	#PU00QYPR	13	18	19.910959	2907	closed	
3281627	#YQYCG2YV	5	6	19.973856	3056	inviteOnly	
1027702	#2P8G2PQ9J	12	17	19.972789	2936	inviteOnly	
1923551	#22UR8GGUV	10	11	20.031250	3005	inviteOnly	
991464	#228CJ8RGU	5	14	18.868293	3868	inviteOnly	
198816	#2QVVRGCLQ	11	5	20.046667	3007	closed	
937598	#9JL8GY9L	7	19	18.231156	3628	inviteOnly	
2499672	#LGC2PJ8P	11	17	18.755208	3601	inviteOnly	
1899511	#9VY8ULUL	13	18	18.543689	3820	inviteOnly	
235632	#9C2JQG9G	14	27	18.388601	3549	inviteOnly	

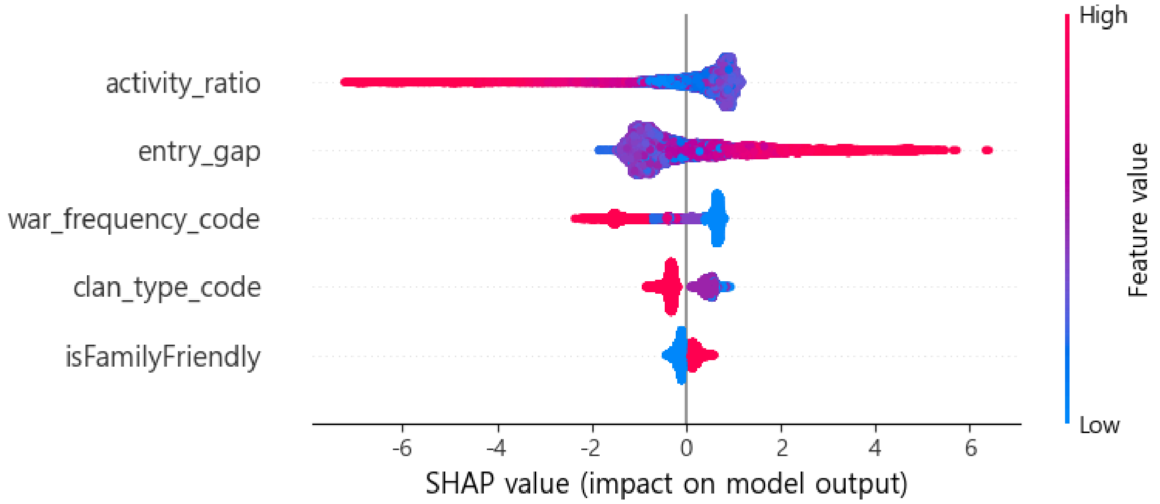
In [58]:

```
import shap

# 1. SHAP Explainer 생성 (Tree 모델 전용)
explainer = shap.TreeExplainer(lgbm_v2)
shap_values = explainer.shap_values(X_test_v2)

# 버전 호환성 체크 (SHAP 결과가 리스트일 경우 [1] 선택)
if isinstance(shap_values, list):
    shap_val = shap_values[1] # Class 1 (생존)에 대한 기여도
else:
    shap_val = shap_values

print("SHAP 값 계산 완료!")

# 2. 어떤 변수가 전반적으로 중요한가? (Summary Plot)
plt.figure(figsize=(10, 6))
shap.summary_plot(shap_val, X_test_v2, plot_type="bar")
plt.show()

# 3. 변수 값이 높을수록 좋은가, 나쁜가? (Beeswarm Plot)
plt.figure(figsize=(10, 6))
shap.summary_plot(shap_val, X_test_v2)
plt.show()
```

SHAP 값 계산 완료!

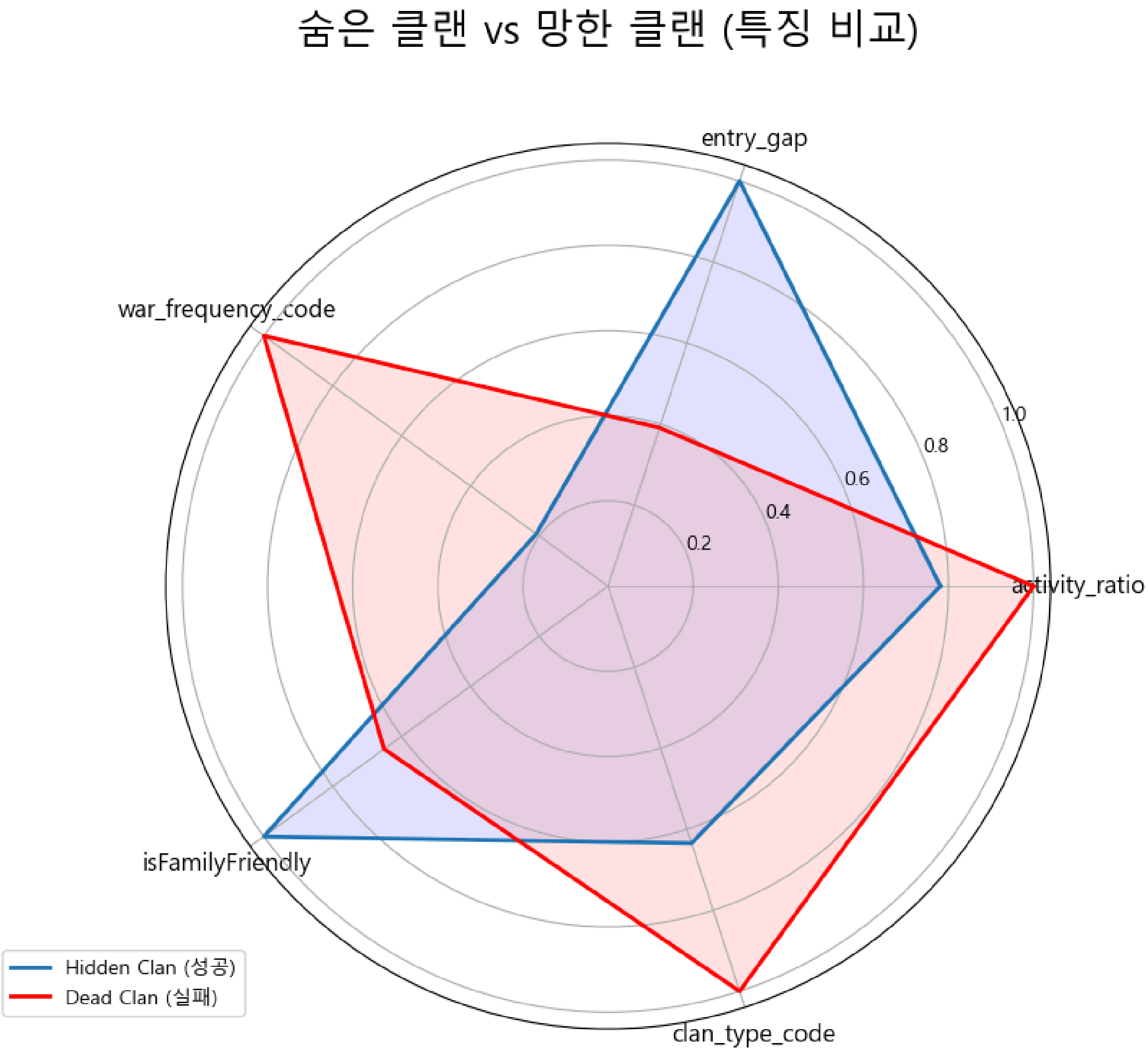
```
c:\Users\tw2ps\AppData\Local\Programs\Python\Python312\Lib\site-packages\shap\explainers\_tree.py:587: UserWarning: LightGBM binary classifier with TreeExplainer shap values output has changed to a list of ndarray
  warnings.warn(
```





In [45]:
```python
import numpy as np
from math import pi

# 1. 비교군 데이터 준비
# (1) Hidden Gems (우리가 찾는 숨은 클랜) 평균
gem_stats = hidden_gems[engineered_features_v2].mean()

# (2) Dead Clans (비슷한 규모인데 망한 클랜들) 평균
# 조건: 멤버 5~15명인데 생존확률 하위 20%인 애들
dead_clans = target_small_clans[target_small_clans['success_prob'] < 0.2]
dead_stats = dead_clans[engineered_features_v2].mean()

# 2. 데이터 정규화 (MinMax Scaling) - 그래프 예쁘게 그리기 위함
merged_stats = pd.concat([gem_stats, dead_stats], axis=1)
merged_stats.columns = ['Hidden Gem', 'Dead Clan']
# 0~1 사이로 변환
normalized_stats = merged_stats.div(merged_stats.max(axis=1), axis=0)

# 3. 레이더 차트 그리기
def make_radar_chart(df, title):
    categories = df.index.tolist()
    N = len(categories)

    # 각도 계산
```

```
    angles = [n / float(N) * 2 * pi for n in range(N)]
    angles += angles[:1]

    fig, ax = plt.subplots(figsize=(8, 8), subplot_kw=dict(polar=True))

    # 첫 번째: Hidden Gem (파란색)
    values = df['Hidden Gem'].tolist()
    values += values[:1]
    ax.plot(angles, values, linewidth=2, linestyle='solid', label='Hidden Clan (
    ax.fill(angles, values, 'b', alpha=0.1)

    # 두 번째: Dead Clan (빨간색)
    values = df['Dead Clan'].tolist()
    values += values[:1]
    ax.plot(angles, values, linewidth=2, linestyle='solid', color='red', label='
    ax.fill(angles, values, 'r', alpha=0.1)

    # 라벨 설정
    plt.xticks(angles[:-1], categories, size=12)
    plt.title(title, size=20, y=1.1)
    plt.legend(loc='upper right', bbox_to_anchor=(0.1, 0.1))
    plt.show()

make_radar_chart(normalized_stats, "숨은 클랜 vs 망한 클랜 (특징 비교)")
print("수치 비교:\n", merged_stats)
```



```
수치 비교:
                      Hidden Gem   Dead Clan
activity_ratio         17.740228   22.713712
entry_gap            1324.800278  518.223451
war_frequency_code      0.711557    3.410109
isFamilyFriendly        0.450836    0.293406
clan_type_code          1.156838    1.822812
```

In [59]:

```python
import joblib

# 1. 모델 저장 (pickle 형태)
joblib.dump(lgbm_v2, 'clan_retention_model.pkl')

# 2. LabelEncoder도 저장 (war_frequency 변환에 필요)
joblib.dump(le, 'war_frequency_encoder.pkl')
# clan_type 인코더도 저장
joblib.dump(le_clan_type, 'clan_type_encoder.pkl')

print("모델 및 인코더 저장 완료!")
print("📁 생성된 파일: clan_retention_model.pkl, war_frequency_encoder.pkl, clan_
```

```
모델 및 인코더 저장 완료!
📁 생성된 파일: clan_retention_model.pkl, war_frequency_encoder.pkl, clan_type_en
coder.pkl
```