分布式系统与Raft面试题

> 基于简历：理解Raft共识算法、分布式一致性、故障恢复、日志复制等核心机制，有完整的分布式KV存储实现经验

一、分布式系统基础

Q1: CAP理论是什么？如何权衡？

CAP三个特性：

C - Consistency（一致性）：

- 所有节点在同一时间看到相同的数据
- 读操作总是返回最新写入的数据

A - Availability（可用性）：

- 每个请求都能得到响应（成功或失败）
- 系统持续可用，不会长时间无响应

P - Partition Tolerance（分区容错性）：

- 网络分区（部分节点无法通信）时，系统仍能继续工作

CAP定理：

- 最多同时满足两个
- 网络分区不可避免，**实际上是在C和A之间权衡**

权衡策略：

CP系统（牺牲可用性）：

- 保证一致性，分区时部分节点不可用
- 例如：Raft、ZooKeeper
- 场景：金融系统、配置中心

AP系统（牺牲一致性）：

- 保证可用性，分区时允许数据不一致
- 例如：Cassandra、DynamoDB
- 场景：社交网络、DNS

CA系统（理论上）：

- 单机系统才能保证
- 分布式系统必须容忍分区

与你项目的联系： "我的Raft KV存储选择CP，牺牲可用性保证强一致性。当网络分区导致无法达成多数派时，系统拒绝服务，但保证数据一致。"

Q2: 分布式一致性算法有哪些?

1. 2PC（两阶段提交）

阶段1：准备（Prepare）

```
协调者 → 所有参与者: 准备提交
参与者: 锁定资源，返回YES/NO
```

阶段2：提交（Commit）

```
if (所有参与者都YES):
    协调者 → 所有参与者: COMMIT
else:
    协调者 → 所有参与者: ABORT
```

问题：

- 阻塞：参与者等待协调者指令期间锁定资源
- 单点故障：协调者挂了，所有参与者阻塞

2. 3PC（三阶段提交）

- 增加超时机制，减少阻塞
- 仍有一致性问题

3. Paxos

- 理论完备，但难以理解和实现
- 分为Basic Paxos和Multi-Paxos

4. Raft

- Paxos的易理解版本
- 分为Leader选举、日志复制、安全性保证
- 工程实践广泛（etcd、consul）

5. ZAB（ZooKeeper Atomic Broadcast）

- ZooKeeper使用
- 类似Raft，但有差异

对比：

算法	优点	缺点	使用

算法	优点	缺点	使用
2PC/3PC	简单	阻塞、单点故障	数据库事务
Paxos	理论完备	难理解和实现	Chubby
Raft	易理解、易实现	性能略逊	etcd、consul

二、Raft算法核心

Q3: Raft的三个核心模块是什么?

1. Leader Election（领导者选举）

- 选出一个Leader负责处理客户端请求
- 保证同一term只有一个Leader

2. Log Replication（日志复制）

- Leader接收客户端请求，复制到Follower
- 过半确认后提交

3. Safety（安全性保证）

- Leader Completeness：已提交的日志不会丢失
- State Machine Safety：相同index的日志，命令相同

Q4: Leader选举的详细流程?

节点状态：

- **Follower**：跟随者，被动接收RPC
- **Candidate**：候选人，发起选举
- **Leader**：领导者，处理客户端请求

选举流程：

1. 触发选举

```
Follower: 心跳超时(随机150-300ms)
    ↓
转为Candidate
```

2. 发起选举

```
void startElection() {
    currentTerm_++;           // term自增
    votedFor_ = me_;          // 给自己投票
    state_ = Candidate;
```

```
        voteCount_ = 1;

        // 并行发送RequestVote RPC到所有节点
        for (int peer : peers_) {
            sendRequestVote(peer);
        }
    }
```

3. 投票规则

```
bool handleRequestVote(RequestVoteArgs args) {
    // 1. candidate的term >= 自己的term
    if (args.term < currentTerm_) {
        return false;
    }

    // 2. 自己在这个term还没投过票，或者已经投给了candidate
    if (votedFor_ != -1 && votedFor_ != args.candidateId) {
        return false;
    }

    // 3. candidate的日志至少和自己一样新
    if (args.lastLogTerm < lastLogTerm_ ||
        (args.lastLogTerm == lastLogTerm_ && args.lastLogIndex <
lastLogIndex_)) {
        return false;
    }

    votedFor_ = args.candidateId;
    return true;
}
```

日志新旧比较：

```
日志更新的定义：
1. 先比较lastLogTerm，term大的更新
2. term相同，比较lastLogIndex，index大的更新
```

4. 成为Leader

```
if (voteCount > peers.size() / 2):
    state = Leader
    立即发送心跳（空AppendEntries）
```

5. 随机超时避免选票分裂

```
如果两个节点同时超时：
- 各自给自己投票
- 都得不到多数票
- 选举失败，重新随机超时
- 下一轮大概率错开
```

与你项目的联系： "我的实现中，心跳超时设置150-300ms随机值。测试中，一般500ms内就能选出Leader，很少出现多轮选举。"

Q5: 日志复制的详细流程?

日志结构：

```
struct LogEntry {
    int term;       // 任期号
    Command cmd;    // 命令（如Put("key", "value")）
    int index;      // 日志索引
};
```

复制流程：

1. 客户端请求

```
Client → Leader: Put("key1", "value1")
```

2. Leader追加日志

```
void handleClientRequest(Command cmd) {
    LogEntry entry;
    entry.term = currentTerm_;
    entry.cmd = cmd;
    entry.index = log_.size();
    log_.push_back(entry);  // 追加到本地日志

    // 并行复制到所有Follower
    for (int peer : peers_) {
        replicateLog(peer);
    }
}
```

3. 发送AppendEntries RPC

```

```

```
struct AppendEntriesArgs {
    int term;           // Leader的term
    int prevLogIndex;   // 前一条日志的index
    int prevLogTerm;    // 前一条日志的term
    LogEntry[] entries; // 要复制的日志
    int leaderCommit;   // Leader的commitIndex
};
```

4. Follower一致性检查

```
bool handleAppendEntries(AppendEntriesArgs args) {
    // 1. term检查
    if (args.term < currentTerm_) {
        return false;
    }

    // 2. 一致性检查：prevLogIndex位置的term是否匹配
    if (log_[args.prevLogIndex].term != args.prevLogTerm) {
        return false;  // 不匹配，返回失败
    }

    // 3. 追加日志
    log_.insert(log_.begin() + args.prevLogIndex + 1,
                args.entries.begin(), args.entries.end());

    // 4. 更新commitIndex
    if (args.leaderCommit > commitIndex_) {
        commitIndex_ = min(args.leaderCommit, log_.size() - 1);
    }

    return true;
}
```

5. Leader提交

```
// Leader统计复制成功的节点
int successCount = 1;  // 包括自己
for (Follower f : followers_) {
    if (f.matchIndex_ >= logIndex) {
        successCount++;
    }
}

// 过半确认，提交
if (successCount > peers_.size() / 2) {
    commitIndex_ = logIndex;
    applyToStateMachine();  // 应用到状态机
    // 返回客户端成功
}
```

6. 日志冲突处理

```
Leader: [1][2][3][4][5]
Follower: [1][2][9]

AppendEntries(prevLogIndex=3, prevLogTerm=3):
Follower检查: log_[3].term != 3
返回失败

Leader递减nextIndex，重试:
AppendEntries(prevLogIndex=2, prevLogTerm=2):
成功匹配，从index=3开始覆盖
Follower: [1][2][3][4][5]
```

Q6: 日志匹配特性（Log Matching Property）是什么?

日志匹配特性： 如果两个节点的日志在某个索引位置的term相同，则：

1. 该位置的命令一定相同
2. 该位置之前的所有日志都相同

为什么能保证?

1. Leader只在一个term追加日志

- 一个term内，最多只有一个Leader
- Leader不会覆盖或删除自己的日志
- 所以同一个(index, term)位置，命令唯一

2. AppendEntries的一致性检查

```
Leader发送: (prevLogIndex, prevLogTerm, entries)
Follower检查: 我在prevLogIndex位置的term是否等于prevLogTerm
如果不等: 返回失败，Leader回退
如果相等: 追加entries

通过这个检查，保证prevLogIndex之前的日志都匹配
```

示例:

```
节点A: [1,1][2,1][3,2][4,2]
节点B: [1,1][2,1][3,2][4,2]

因为index=3的term都是2，所以：
```

```
- index=3的命令相同
- index=0,1,2的日志也相同
```

Q7: Raft如何保证已提交的日志不丢失?

Leader Completeness Property： 如果一条日志在某个term被提交，那么所有后续term的Leader都包含这条日志。

如何保证?

1. 投票限制

```
// Candidate发送RequestVote时带上：
lastLogIndex, lastLogTerm

// Follower投票前检查：
if (candidate的日志没有自己新) {
    拒绝投票
}
```

日志新旧比较：

```
if (candidateLastLogTerm > myLastLogTerm) {
    candidate更新
} else if (candidateLastLogTerm == myLastLogTerm) {
    if (candidateLastLogIndex >= myLastLogIndex) {
        candidate更新
    }
}
```

2. 为什么有效?

```
假设日志L在term T被提交：
- L被复制到了多数节点（包括Leader）
- 任何后续Leader都需要获得多数票
- 多数集合必有交集（鸽巢原理）
- 所以后续Leader必然从"拥有L的节点"中选出
- 而拥有L的节点不会投票给日志比自己旧的节点
- 因此后续Leader必然包含L
```

3. 新Leader不能直接提交旧term的日志

```
节点1(Leader): [1,1][2,1][3,2]
节点2: [1,1][2,1]
```

```
节点3: [1,1][2,1][3,2]
节点4: [1,1]
节点5: [1,1]

index=3的日志已复制到3个节点，但还未提交
节点1宕机，节点5当选Leader(term=3)
节点5可能覆盖index=3的日志！

解决：新Leader只能通过提交当前term的日志，间接提交旧term的日志
```

三、Raft工程实践

Q8: 快照（Snapshot）机制如何实现?

为什么需要快照?

- 日志无限增长，占用大量空间
- 节点重启时，replay所有日志很慢

快照内容:

```
struct Snapshot {
    int lastIncludedIndex;  // 快照包含的最后一条日志索引
    int lastIncludedTerm;   // 最后一条日志的term
    byte[] data;            // 状态机数据（KV数据）
};
```

生成快照:

```
void createSnapshot() {
    if (log_.size() < snapshotThreshold_) {
        return;  // 日志不够多，不需要快照
    }

    Snapshot snap;
    snap.lastIncludedIndex = commitIndex_;
    snap.lastIncludedTerm = log_[commitIndex_].term;
    snap.data = serializeStateMachine();  // 序列化KV数据

    saveSnapshot(snap);

    // 删除快照包含的日志
    log_.erase(log_.begin(), log_.begin() + commitIndex_ + 1);
}
```

节点重启恢复:

```
void recover() {
    Snapshot snap = loadSnapshot();
    if (snap != null) {
        lastIncludedIndex_ = snap.lastIncludedIndex;
        lastIncludedTerm_ = snap.lastIncludedTerm;
        deserializeStateMachine(snap.data);  // 恢复KV数据
    }

    // Replay快照之后的日志
    for (LogEntry entry : log_) {
        applyToStateMachine(entry.cmd);
    }
}
```

Follower落后太多：

```
Leader: [快照][100][101][102]
Follower: [1][2]...[50]  ← 落后太多，Leader没有早期日志

Leader发送InstallSnapshot RPC:
- 传输整个快照
- Follower接收后，丢弃所有日志，应用快照
```

与你项目的联系： "我的实现中，日志超过10000条时生成快照。快照包含跳表的所有KV数据，序列化后保存到磁盘。"

Q9: RPC失败和超时如何处理?

场景：

1. 网络丢包：RPC请求或响应丢失
2. 节点宕机：无响应
3. 网络分区：部分节点无法通信

处理策略：

1. 超时重试

```
void replicateLog(int peer) {
    while (true) {
        AppendEntriesArgs args = buildArgs(peer);
        AppendEntriesReply reply;

        bool ok = sendAppendEntries(peer, args, reply, timeout=1s);

        if (ok && reply.success) {
            matchIndex_[peer] = args.prevLogIndex + args.entries.size();
```

```
                break;
            }

            if (!ok) {
                // 超时或网络错误，重试
                continue;
            }

            if (!reply.success) {
                // 日志不匹配，回退nextIndex重试
                nextIndex_[peer]--;
            }
        }
    }
```

2. 幂等性

```
同一个AppendEntries可能发送多次（重试）
Follower通过(prevLogIndex, prevLogTerm)判断是否重复
重复的RPC，处理结果相同（幂等）
```

3. 不等待慢节点

```
Leader发送AppendEntries到所有Follower
只要多数节点响应，就可以提交
慢节点或故障节点不影响整体进度
```

4. 定期心跳

```
Leader定期（50ms）发送心跳（空AppendEntries）
作用：
- 维持Leader地位
- 防止Follower超时发起选举
- 更新Follower的commitIndex
```

与你项目的联系： "我的实现中，RPC超时设置为1秒。超时后不重试，依赖下一次心跳或日志复制。这样实现简单，但对网络抖动敏感，可以改进。"

Q10: 如何测试Raft的正确性?

测试场景：

1. Leader选举

```
TEST(RaftTest, LeaderElection) {
    // 1. 启动3节点集群
    cluster = createCluster(3);

    // 2. 等待选举
    sleep(1s);

    // 3. 检查有且只有一个Leader
    assert(countLeaders() == 1);

    // 4. kill Leader
    killLeader();

    // 5. 等待重新选举
    sleep(1s);

    // 6. 检查新Leader产生
    assert(countLeaders() == 1);
}
```

2. 日志复制

```
TEST(RaftTest, LogReplication) {
    cluster = createCluster(3);

    // 提交命令
    leader.propose("Put('k1', 'v1')");
    sleep(100ms);

    // 检查所有节点的日志一致
    for (node : cluster) {
        assert(node.getLog() == expectedLog);
    }
}
```

3. 网络分区

```
TEST(RaftTest, NetworkPartition) {
    cluster = createCluster(5);  // A B C D E

    // 分区：{A, B} | {C, D, E}
    partition(A, B);

    // 多数分区(C,D,E)可以继续工作
    C.propose("Put('k1', 'v1')");
    assert(committed);

    // 少数分区(A,B)无法提交
    A.propose("Put('k2', 'v2')");
```

```
    assert(!committed);

    // 恢复分区
    healPartition();
    sleep(1s);

    // 检查一致性
    assertAllNodesSame();
}
```

4. 节点崩溃恢复

```
TEST(RaftTest, CrashRecovery) {
    cluster = createCluster(3);

    // 提交一些命令
    leader.propose("Put('k1', 'v1')");
    leader.propose("Put('k2', 'v2')");

    // 节点2崩溃
    node2.crash();

    // 继续提交
    leader.propose("Put('k3', 'v3')");

    // 节点2恢复
    node2.restart();
    sleep(1s);

    // 检查node2追上日志
    assert(node2.getLog() == leader.getLog());
}
```

5. Jepsen测试

- 注入故障：网络分区、节点崩溃、时钟漂移
- 并发操作：多客户端读写
- 检查一致性：验证线性一致性

与你项目的联系： "我实现了20多个测试用例，覆盖选举、日志复制、节点故障等场景。但没有用Jepsen这种专业工具，这是需要改进的地方。"

四、面试技巧

画图说明

1. Leader选举时序图

```
时间  Follower1      Follower2      Follower3
1     超时
2     Term++
3     RequestVote →
4                     投票 ← Yes
5                                    投票 ← Yes
6     成为Leader
7     心跳 →
```

2. 日志复制流程

```
Client → Leader: Put("k1", "v1")
Leader: 追加本地日志
Leader → Follower: AppendEntries
Follower: 追加日志，返回Success
Leader: 过半确认，提交
Leader → Client: Success
```

结合项目经验

好的回答： "Raft的Leader选举通过随机超时避免选票分裂。在我的实现中，我设置150-300ms的随机超时。测试时发现，如果超时范围太小（如145-155ms），偶尔会出现多轮选举。调整到150-300ms后，基本都能一次选出Leader。"

深入追问准备

Q: "如果网络延迟很大，Raft性能会怎样？"

A: "性能会下降。每次日志复制需要一个RTT，延迟大会影响吞吐。优化方法：

1. 批处理：积累多个命令一起复制
2. 流水线：不等前一个ACK，继续发送下一个
3. Multi-Raft：分片，每个分片独立Raft组"

快速复习清单

CAP理论

- [ ] CAP三个特性
- [ ] CP vs AP的权衡

Raft核心

- [ ] Leader选举流程
- [ ] 日志复制流程
- [ ] 日志匹配特性
- [ ] Leader Completeness

工程实践

- [ ] 快照机制
- [ ] RPC超时处理
- [ ] 测试方法

关键数字

- [ ] 心跳超时：150-300ms随机
- [ ] 心跳间隔：50ms左右
- [ ] 快照阈值：根据场景（我用10k条日志）

记住：原理 + 实现细节 + 遇到的问题 = 完整回答