

SD

Fundamentals · most asked

System Design interview questions

28 questions · 6 easy · 15 medium · 7 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

Vertical vs horizontal scaling?

Easy



#scalability

Vertical scaling (scale up): a bigger machine, more CPU, RAM or faster disks. Simple, no code changes, but there is a hardware ceiling, it gets expensive, and the box is still a single point of failure.

Horizontal scaling (scale out): add more machines behind a load balancer. Near-unlimited capacity and built-in redundancy, but the app must be **stateless** (sessions in Redis, files in S3) and you take on distributed-system problems: consistency, coordination, network failures.

In practice: scale the stateless app tier horizontally, scale the database vertically as long as you can, then add read replicas and sharding.

? Likely follow-up: What makes a service hard to scale horizontally?

2

What does a load balancer do? Which algorithms are common?

Easy



#load balancing #scalability

A load balancer spreads requests across healthy servers, runs **health checks** to remove dead ones, and often terminates TLS.

- **L4** (TCP/UDP) balancers route by IP and port: very fast, content-blind
- **L7** (HTTP) balancers can route by path, header or cookie

Algorithms:

- **round robin** (optionally weighted): simple, good for equal servers and short requests
- **least connections**: better when request durations vary
- **IP hash / consistent hashing**: the same client goes to the same server (sticky), useful for caches

- **random of two choices:** pick two at random, send to the less loaded

The balancer itself must be redundant (active-passive pair, or a managed cloud LB), or it becomes the single point of failure.

3

Explain caching strategies: cache-aside, write-through, write-back.

Medium 🔥🔥 #caching

- **Cache-aside (lazy loading):** the app reads the cache; on a miss it reads the DB and fills the cache. On write it updates the DB and **deletes** the cache key. Most common; only hot data is cached, but the first read is slow and data can be briefly stale.
- **Read-through:** same idea, but the cache library loads from the DB itself.
- **Write-through:** every write goes to the cache and the DB synchronously. Cache is always fresh; writes are slower.
- **Write-back (write-behind):** write to the cache only and flush to the DB later in batches. Very fast writes, but data is lost if the cache dies before flushing.

Pick by read/write ratio and how much staleness or data loss you can tolerate. Always set a **TTL** as a safety net.

Example

```
const db = new Map([[ 'user:1', { name: 'Asha' } ]])
const cache = new Map()
function getUser(key) {
  if (cache.has(key)) return { from: 'cache', v: cache.get(key) }
  const v = db.get(key) // miss: load from DB
  cache.set(key, v) // and populate
  return { from: 'db', v }
}
function updateUser(key, v) {
  db.set(key, v)
  cache.delete(key) // invalidate, don't update
}
console.log(getUser('user:1').from) // db
console.log(getUser('user:1').from) // cache
updateUser('user:1', { name: 'Ben' })
console.log(getUser('user:1')) // db, fresh value
```

🔗 Likely follow-up: Why delete the cache key on write instead of updating it?

4

How do cache eviction and invalidation work? Where can you cache?

Medium



#caching

Eviction decides what to drop when the cache is full:

- **LRU** (least recently used): the usual default
- **LFU** (least frequently used): keeps long-term popular items
- **TTL**: entries expire after a fixed time

Invalidation keeps cached data correct: TTLs, delete-on-write, or events (CDC) that purge keys. Watch for the **thundering herd / cache stampede**: a hot key expires and thousands of requests hit the DB at once. Fix with request coalescing, a lock, or early random refresh.

Where to cache, from client to server: browser (HTTP `Cache-Control`), CDN edge, API gateway, `in-process memory`, distributed cache (Redis, Memcached), and the DB's own buffer cache.

5

What is a CDN and how does it work?

Easy



#caching #cdn

A **Content Delivery Network** is a set of edge servers around the world that `cache content close to users`. DNS or anycast routes a user to the nearest edge; on a miss, the edge fetches from your **origin** and caches it per `Cache-Control` headers.

Benefits: lower latency, far less load and bandwidth on the origin, absorbs traffic spikes and DDoS.

- **Pull CDN**: fetches lazily on first request (most common)
- **Push CDN**: you upload content in advance (large, rarely changing files)

For static assets use **content-hashed filenames** (`app.3f9a.js`) with a long max-age, so a deploy never needs a purge. Modern CDNs also cache API responses and run edge functions.

6

How does database replication work? What are read replicas?

Medium



#databases #replication

Leader-follower (primary-replica): all writes go to the leader, which streams its change log to followers. Reads can be spread over **read replicas**, which scales read-heavy workloads and gives failover targets.

- **Synchronous** replication: the write waits for a replica to confirm; no data loss on failover, but slower
- **Asynchronous:** fast, but replicas lag, so you get **replication lag** issues like a user not seeing their own just-saved post

Fixes for lag: read-your-own-writes (read from the leader for a few seconds after a write, or for that user's own data).

Multi-leader and **leaderless** (Cassandra, Dynamo-style quorums) allow writes in several places but must resolve conflicts.

? **Likely follow-up:** What happens if the leader fails?

7

What is sharding? How do you choose a shard key?

Hard



#databases #sharding

Sharding (horizontal partitioning) splits one table's rows across several databases, so each holds a subset. It scales **writes** and storage, which replicas cannot.

Strategies:

- **range** (by date, A-M / N-Z): easy range scans, but risks hot spots
- **hash** of the key: even spread, but range queries hit every shard
- **directory/lookup:** a service maps keys to shards, flexible but one more component

A good **shard key** has high cardinality, spreads load evenly and **keeps most queries on a single shard** (e.g. `user_id` for a per-user app).

Costs: cross-shard joins and transactions are hard, rebalancing is painful, and celebrity keys create hot shards. Shard only after caching, replicas and vertical scaling are exhausted.

🔗 Likely follow-up: How do you handle a hot shard caused by one celebrity user?

8

What is consistent hashing and why is it used?

Hard



#distributed systems #sharding

With naive $\text{hash}(\text{key}) \% N$, changing the number of servers N remaps **almost every key**, which empties a cache or forces a huge data migration.

Consistent hashing places servers and keys on a ring (0 to 2^{32}). A key belongs to the **first server clockwise** from its hash. Adding or removing a server moves only about $1/N$ of the keys, those between it and its neighbour.

Virtual nodes: each server appears at many points on the ring, which evens out the load and spreads a failed server's keys over many others.

Used in DynamoDB, Cassandra, memcached clients and CDNs.

Example

```
const hash = (s) => { // FNV-1a + mixing, 32-bit
  let h = 2166136261
  for (const c of s) h = Math.imul(h ^ c.charCodeAt(0), 16777619)
  h ^= h >>> 16; h = Math.imul(h, 0x85ebca6b); h ^= h >>> 13
  return h >>> 0
}

class Ring {
  constructor(nodes, vnodes = 100) {
    this.points = []
    for (const n of nodes) for (let i = 0; i < vnodes; i++)
    this.points.push([hash(n + '#' + i), n])
    this.points.sort((a, b) => a[0] - b[0])
  }
  get(key) {
    const h = hash(key)
    const p = this.points.find(([pos]) => pos >= h) ?? this.points[0] //
    wrap around
    return p[1]
  }
}

const keys = Array.from({ length: 1000 }, (_, i) => 'user' + i)
const before = new Ring(['A', 'B', 'C'])
const after = new Ring(['A', 'B', 'C', 'D'])
const moved = keys.filter((k) => before.get(k) !== after.get(k)).length
console.log('keys moved after adding D:', moved, 'of', keys.length) //
~1/4, all to D
```

9

Explain the CAP theorem and PACELC.

Medium



#distributed systems #consistency

CAP: in a distributed data store, when a **network partition** happens you must choose between:

- **Consistency:** every read sees the latest write (or gets an error)
- **Availability:** every request gets a non-error response, possibly stale

Partitions are unavoidable, so the real choice is **CP or AP during a partition**. Banking ledgers, inventory and locks lean CP (etcd, ZooKeeper, HBase); social feeds and carts lean AP (Cassandra, DynamoDB default).

PACELC adds the normal case: if Partition, choose A or C; **Else**, choose **Latency or Consistency**. Even without failures, synchronous replication costs latency. Cassandra is

PA/EL; a classic RDBMS cluster is PC/EC.

 **Likely follow-up:** Is MongoDB CP or AP?

10

Strong vs eventual consistency, and models in between?

Hard



#consistency #distributed systems

- **Strong (linearizable):** once a write succeeds, every reader sees it. Simple to reason about, but needs coordination (consensus, quorums), so it is slower and less available.
- **Eventual:** replicas converge if writes stop; reads may be stale for a while. Fast and highly available.

In between:

- **Read-your-writes:** a user always sees their own updates
- **Monotonic reads:** you never see data go back in time
- **Causal:** if A caused B (comment after post), everyone sees A before B

In quorum systems, $R + W > N$ gives you overlapping read and write sets, so reads see the latest write. Choose per feature: money needs strong, like counts are fine eventual.

11

SQL vs NoSQL: how do you choose?

Easy



#databases

SQL (Postgres, MySQL): fixed schema, joins, **ACID** transactions, powerful ad-hoc queries. Choose it for relational data and correctness: payments, orders, users. Scales with replicas, partitioning and, eventually, sharding.

NoSQL families:

- **key-value** (Redis, DynamoDB): simple, extremely fast lookups
- **document** (MongoDB): flexible JSON, data read together stored together
- **wide-column** (Cassandra): huge write throughput, time series
- **graph** (Neo4j): relationships, recommendations

Choose by **access patterns**, consistency needs, scale and team skill. "Postgres by default" is a reasonable answer; add a NoSQL store where a specific pattern (massive writes, cache, flexible schema) demands it.

12

Why use a message queue? Kafka vs RabbitMQ? Queue vs pub/sub?

Medium



#messaging #async

Queues **decouple** producers from consumers: absorb traffic spikes, do slow work asynchronously (emails, video encoding), retry failures, and let services fail independently.

- **Queue (point-to-point)**: each message is processed by **one** consumer of a pool
- **Pub/sub**: each message goes to **every** subscriber group

RabbitMQ: a classic broker with flexible routing (exchanges), per-message acks, and messages deleted once consumed. Great for task queues.

Kafka: a **distributed, append-only log**. Messages are kept for days, consumers track their own offsets and can replay, ordering is guaranteed per partition, and throughput is huge. Great for event streaming, analytics and event sourcing.

Delivery is usually **at-least-once**, so consumers must be idempotent.

? **Likely follow-up**: How do you guarantee ordering in Kafka?

13

Microservices vs monolith?

Medium



#architecture #microservices

Monolith: one deployable. Simple to develop, test, debug and deploy; in-process calls are fast and transactions easy. It hurts when a large team steps on each other and everything must scale and deploy together.

Microservices: small services owned by teams, each with its own database, deployed independently and scaled separately, possibly in different languages. The cost: network calls and partial failures, distributed transactions, eventual consistency, observability, and a lot of DevOps.

A good answer: **start with a modular monolith**, with clear module boundaries, and extract services when there is a real reason (team size, very different scaling needs, independent release cadence). Avoid a "distributed monolith" where services must deploy together.

14

What is an API gateway?

Easy



#architecture #microservices

A single entry point in front of many backend services. It handles **cross-cutting concerns** so services do not each reimplement them:

- routing (`/orders` to the order service)
- authentication, token validation
- rate limiting and quotas
- TLS termination, request/response transformation
- aggregating several service calls into one response (**Backend for Frontend** pattern)
- logging, metrics, caching

Examples: Kong, AWS API Gateway, Envoy, Nginx. Downsides: an extra hop and a component that must be **highly available** and must not become a home for business logic.

15

What are the common rate limiting algorithms?

Medium



#rate limiting

- **Fixed window counter**: count per minute bucket. Simple, but allows bursts of 2x at window edges.
- **Sliding window log**: store each request timestamp; exact but memory-heavy.
- **Sliding window counter**: blend the current and previous window counts by overlap; cheap and accurate enough.
- **Token bucket**: a bucket of capacity B refills at rate R; each request takes a token. **Allows short bursts** while enforcing an average rate. Most popular (AWS, Stripe).
- **Leaky bucket**: requests queue and drain at a constant rate; smooths output.

Return `429 Too Many Requests` with a `Retry-After` header.

Example

```
function tokenBucket(capacity, refillPerSec) {
  let tokens = capacity
  let last = Date.now()
  return function allow(now = Date.now()) {
    tokens = Math.min(capacity, tokens + ((now - last) / 1000) *
refillPerSec)
    last = now
    if (tokens >= 1) { tokens -= 1; return true }
    return false
  }
}

const allow = tokenBucket(3, 1) // burst 3, then 1 per second
const t0 = Date.now()
console.log([0, 0, 0, 0].map(() => allow(t0))) // true true true false
console.log(allow(t0 + 1000)) // true: one token
refilled
```

16

Design a distributed rate limiter.

Hard



#design #rate limiting

Requirements: limit per user/IP/API key (e.g. 100 req/min), low latency (under a few ms), works across many API servers, configurable rules, fail open if the limiter is down.

Where: middleware in the API gateway, before the services.

Data model: Redis key `rl:{userId}:{window}` with a counter, or token-bucket state `{tokens, lastRefill}`, with a TTL.

Core flow: on each request, run an **atomic Redis operation** (a Lua script, or `INCR` + `EXPIRE`) to update and check the count; allowed requests pass, others get `429` with `X-RateLimit-Remaining` and `Retry-After`.

Scaling: Redis Cluster sharded by key; rules cached locally in each gateway.

Trade-offs: atomic scripts avoid race conditions; local in-memory pre-limits cut Redis load at the cost of precision; fail open keeps the API up but unprotected.

17

What is idempotency, and how should clients retry failed requests?

Medium



#reliability #api

An operation is **idempotent** if doing it twice has the same effect as once. **GET**, **PUT**, **DELETE** should be; **POST** usually is not. It matters because with timeouts you cannot tell if a request failed or only its response was lost, so retries can double-charge a card.

Make writes idempotent with an **idempotency key**: the client sends a unique **Idempotency-Key** header; the server stores the key with the result and returns the saved result for repeats.

Retries: only for transient errors (timeouts, 503), with **exponential backoff plus jitter** (100ms, 200ms, 400ms... randomised so clients do not retry in sync), a max attempt count, and a **circuit breaker** to stop hammering a dead service.

Example

```
const sleep = (ms) => new Promise((r) => setTimeout(r, ms))
async function retry(fn, { attempts = 4, base = 20 } = {}) {
  for (let i = 0; ; i++) {
    try { return await fn() } catch (e) {
      if (i === attempts - 1) throw e
      const delay = base * 2 ** i * (0.5 + Math.random() / 2) // backoff
      + jitter
      console.log('attempt', i + 1, 'failed, waiting', Math.round(delay),
        'ms')
      await sleep(delay)
    }
  }
}
let calls = 0
const flaky = async () => { if (++calls < 3) throw new Error('503');
  return 'ok' }
retry(flaky).then((r) => console.log('result', r, 'after', calls,
  'calls'))
```

18

How do you handle transactions across microservices? What is a saga?

Hard



#microservices #transactions

Each service owns its database, so one ACID transaction is impossible. **Two-phase commit (2PC)** exists but blocks if the coordinator fails and couples services tightly, so it

is rarely used.

A **saga** is a sequence of local transactions; **each step has a compensating action** that undoes it if a later step fails. Example order flow: create order, reserve stock, charge card. If charging fails: release stock, cancel order.

- **Choreography**: services react to each other's events; no central brain, harder to follow
- **Orchestration**: a coordinator tells each service what to do; easier to reason about

Use the **outbox pattern** to write the DB change and the event atomically, and make every step idempotent.

? Likely follow-up: What is the transactional outbox pattern?

19

What is event-driven architecture?

Medium 🔥🔥 #architecture #messaging

Services communicate by **publishing events** ("OrderPlaced") to a broker, and other services **subscribe** and react, instead of calling each other directly.

Benefits:

- **loose coupling**: the producer does not know who listens; add consumers without changing it
- async processing and natural buffering of spikes
- a history of events enables audit logs, analytics and replay

Costs: eventual consistency, harder debugging (trace IDs needed), duplicate and out-of-order events, and event schema versioning.

Related patterns: **event sourcing** (store events as the source of truth and derive state) and **CQRS** (separate write and read models).

20

WebSockets vs Server-Sent Events vs long polling?

Medium 🔥🔥 #realtime #networking

- **Short polling**: the client asks every few seconds. Simple, wasteful, laggy.
- **Long polling**: the server holds the request open until there is data, then the client immediately reconnects. Works everywhere; extra overhead per message.

- **SSE:** one long HTTP response streaming events server to client only. Auto-reconnect built in, plain HTTP, text only. Great for notifications, live scores, AI token streaming.
- **WebSockets:** a full-duplex persistent connection, both directions, low overhead. Needed for chat, multiplayer games, collaborative editing.

Scaling persistent connections needs sticky sessions or a pub/sub layer (Redis) so any server can push to any user, plus raised file-descriptor limits.

21

Design a URL shortener (like bit.ly).

Medium



#design

Requirements: shorten a URL, redirect fast, optional custom alias and expiry, analytics. Read-heavy (~100:1). Example: 100M new URLs/month, ~40 writes/s, ~4K reads/s.

API: `POST /urls {longUrl}` returns `{code}`; `GET /{code}` returns a **301/302** redirect (302 if you need click analytics).

Data model: `urls(code PK, long_url, user_id, created_at, expires_at)`. A key-value store or sharded SQL both work.

Code generation: 7 chars of base62 gives 3.5 trillion codes. Best: a unique counter ID encoded in base62 (counter ranges handed out per server, or Snowflake IDs). Hashing the URL needs collision checks.

Scaling: cache hot codes in Redis and the CDN, read replicas, async click events to Kafka for analytics.

Trade-offs: sequential codes are guessable (add a shuffle), 301 is cached by browsers so it loses analytics.

Example

```
const ALPHABET =
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
function toBase62(n) {
  let s = ''
  do { s = ALPHABET[n % 62] + s; n = Math.floor(n / 62) } while (n > 0)
  return s
}
function fromBase62(s) {
  return [...s].reduce((n, c) => n * 62 + ALPHABET.indexOf(c), 0)
}
const id = 125_000_000_000
const code = toBase62(id)
console.log(code, fromBase62(code) === id)
console.log('7-char codes:', (62 ** 7).toLocaleString())
```

? Likely follow-up: How do you stop two servers generating the same code?

22

Design a chat application (like WhatsApp).

Hard



#design #realtime

Requirements: 1:1 and group chat, online presence, delivered/read receipts, message history, offline delivery, low latency.

API: WebSocket for send/receive; REST for history (`GET /conversations/{id}/messages?before=`) and media upload (to S3, send the URL).

Data model: `messages(conversation_id, message_id, sender_id, body, created_at)` partitioned by `conversation_id` and sorted by time-ordered ID; a wide-column store like Cassandra suits this write-heavy pattern.

Core components: stateful **chat servers** holding WebSockets; a **session registry (Redis)** mapping user to server; a pub/sub or queue to route messages between servers; push notifications for offline users; presence via heartbeats.

Flow: sender's server persists the message, looks up the recipient's server and forwards; if offline, store and push-notify.

Scaling and trade-offs: large groups fan out on read; ordering per conversation via sequence numbers; end-to-end encryption limits server-side search.

23

Design a news feed (like Twitter/Instagram home timeline).

Hard



#design

Requirements: post content, follow users, see a feed of followed users' posts, newest or ranked, fast loads (under 200ms). Very read-heavy.

API: `POST /posts`, `GET /feed?cursor=` with cursor pagination.

Data model: `posts`, `follows(follower_id, followee_id)`, and a precomputed **feed cache** per user (a list of post IDs in Redis).

Core choice: fan-out

- **Fan-out on write (push):** on post, append the ID to every follower's feed. Reads are instant; terrible for a celebrity with 50M followers.
- **Fan-out on read (pull):** build the feed at read time by merging followees' posts. Cheap writes, slow reads.
- **Hybrid:** push for normal users, pull celebrity posts at read time and merge.

Scaling: async fan-out workers via a queue, post objects cached separately (hydrate IDs), media on a CDN, ranking service for ordering.

? **Likely follow-up:** How would you add ranking instead of pure chronological order?

24

Design a notification system (push, email, SMS).

Medium



#design #messaging

Requirements: send via push (APNs/FCM), email and SMS; respect user preferences and quiet hours; templates; scheduling; at-least-once delivery without spamming duplicates; high throughput.

API: `POST /notifications {userId, type, templateId, data, channels}` for internal services.

Data model: user preferences and device tokens, templates, a `notifications` log with status (queued, sent, failed, read).

Core components: notification service (validate, check preferences, rate limit per user, render template) then **one queue per channel**, with channel workers calling third-party providers. Retries with backoff and a dead-letter queue; idempotency keys stop duplicates.

Scaling: workers scale horizontally per queue; separate queues stop a slow SMS provider from delaying push.

Trade-offs: provider failover vs cost, batching digests vs immediacy, and tracking delivery and open rates via provider webhooks.

25

How do you do back-of-the-envelope estimation?

Medium



#estimation

Estimate scale to justify design choices; round aggressively and state assumptions.

Handy numbers:

- 1 day is about **86,400 s**, round to **100K s**
- 1M requests/day is about **12 req/s**; peak is usually 2-3x average
- memory read ~100ns, SSD read ~100μs, same-datacenter round trip ~0.5ms, cross-continent ~150ms

Example: 100M DAU, each makes 10 reads/day: 1B reads/day, about **10K reads/s average**, ~30K at peak. Each user posts 1 item of 1 KB/day: 100 GB/day, ~36 TB/year, so the data must be sharded and old data tiered.

Interviewers care that you derive QPS, storage and bandwidth and then use them to decide (one DB vs sharding, cache size).

Example

```
const dau = 100e6
const readsPerUser = 10
const qps = (dau * readsPerUser) / 86400
console.log('avg read QPS:', Math.round(qps), 'peak:', Math.round(qps * 3))
const bytesPerPost = 1024
const perDayGB = (dau * bytesPerPost) / 1e9
console.log('storage/day GB:', perDayGB.toFixed(0), '/year TB:',
(perDayGB * 365 / 1e3).toFixed(1))
```

26

How do you design for high availability? What is a single point of failure?

Medium



#reliability

A **single point of failure (SPOF)** is any component whose failure takes the whole system down: one DB, one load balancer, one region, even one engineer with the password.

High availability means **redundancy everywhere** plus automatic failover:

- multiple app instances across **availability zones**
- redundant load balancers, DB with a standby replica and automatic failover
- health checks, timeouts, retries, circuit breakers, graceful degradation (serve cached or partial data)
- multi-region for disaster recovery, with defined **RTO/RPO**

Availability math: 99.9% allows ~8.8 hours of downtime per year, 99.99% ~53 minutes. Components in series multiply (two 99.9% services give 99.8%); redundancy in parallel raises it.

27

What are the three pillars of observability?

Easy



#observability #reliability

- **Logs:** discrete events with context. Structured JSON with a request ID, shipped to a central store (ELK, Loki). Good for "what exactly happened".
- **Metrics:** numeric time series (request rate, error rate, latency percentiles, CPU). Cheap to store, great for dashboards and alerts. Prometheus, Grafana.
- **Traces:** follow **one request across services** as a tree of spans with timings; shows where latency comes from. OpenTelemetry, Jaeger.

Tie them together with a shared **trace ID**. Alert on **SLOs** and user-facing symptoms (p99 latency, error rate), using the RED (rate, errors, duration) or USE (utilisation, saturation, errors) methods.

28

How do you generate unique IDs in a distributed system? Explain Snowflake.

Medium



#distributed systems #ids

Options:

- **DB auto-increment:** simple but a bottleneck and SPOF; leaks counts

- **UUID v4**: no coordination, but 128 bits, random, so poor index locality
- **Ticket server / range allocation**: each node grabs a block of IDs
- **Snowflake** (Twitter): a 64-bit integer made of **41 bits timestamp** (ms since a custom epoch, ~69 years), **10 bits machine ID**, **12 bits sequence** (4096 IDs per ms per machine)

Snowflake IDs are **roughly time-sortable**, fit in a **BIGINT**, and need no coordination at runtime. Watch out for clock moving backwards (wait or refuse). UUID v7 and ULID give similar time-ordering in a UUID format.

Example

```
const EPOCH = 1704067200000n // 2024-01-01
function snowflake(machineId) {
  let lastMs = -1n, seq = 0n
  return () => {
    let now = BigInt(Date.now())
    if (now === lastMs) {
      seq = (seq + 1n) & 4095n
      if (seq === 0n) while (now <= lastMs) now = BigInt(Date.now()) //
4096 used: wait 1 ms
    } else seq = 0n
    lastMs = now
    return ((now - EPOCH) << 22n) | (BigInt(machineId) << 12n) | seq
  }
}
const next = snowflake(7)
const a = next(), b = next()
console.log(a.toString(), b.toString(), b > a)
console.log('machine:', Number((a >> 12n) & 1023n))
```