```
# AdjointTensorMap: lazy adjoint
#-------------------------------------------------------------#
"""
    struct AdjointTensorMap{S<:IndexSpace, N₁, N₂, ...} <: AbstractTensorMap{S,
N₁, N₂}

Specific subtype of [`AbstractTensorMap`](@ref) that is a lazy wrapper for
representing the
adjoint of an instance of [`TensorMap`](@ref).
"""
struct AdjointTensorMap{S<:IndexSpace, N₁, N₂, I<:Sector, A, F₁, F₂} <:
                                                            AbstractTensorMap{S,
N₁, N₂}
    parent::TensorMap{S, N₂, N₁, I, A, F₂, F₁}
end

const AdjointTrivialTensorMap{S<:IndexSpace, N₁, N₂, A<:DenseMatrix} =
    AdjointTensorMap{S, N₁, N₂, Trivial, A, Nothing, Nothing}

# Constructor: construct from taking adjoint of a tensor
Base.adjoint(t::TensorMap) = AdjointTensorMap(t)
Base.adjoint(t::AdjointTensorMap) = t.parent

# Properties
codomain(t::AdjointTensorMap) = domain(t.parent)
domain(t::AdjointTensorMap) = codomain(t.parent)

blocksectors(t::AdjointTensorMap) = blocksectors(t.parent)

storagetype(::Type{<:AdjointTensorMap{<:IndexSpace, N₁, N₂, Trivial, A}}) where
{N₁, N₂, A<:DenseMatrix} = A
storagetype(::Type{<:AdjointTensorMap{<:IndexSpace, N₁, N₂, I, <:SectorDict{I,
A}}}) where {N₁, N₂, I<:Sector, A<:DenseMatrix} = A

dim(t::AdjointTensorMap) = dim(t.parent)

# Indexing
#----------
hasblock(t::AdjointTensorMap, s::Sector) = hasblock(t.parent, s)
block(t::AdjointTensorMap, s::Sector) = block(t.parent, s)'
blocks(t::AdjointTensorMap) = (c=>b' for (c, b) in blocks(t.parent))

fusiontrees(::AdjointTrivialTensorMap) = ((nothing, nothing),)
fusiontrees(t::AdjointTensorMap) = TensorKeyIterator(t.parent.colr, t.parent.rowr)

function Base.getindex(t::AdjointTensorMap{S, N₁, N₂, I},
                       f1::FusionTree{I, N₁}, f2::FusionTree{I, N₂}) where {S,
N₁, N₂, I}
    c = f1.coupled
    @boundscheck begin
        c == f2.coupled || throw(SectorMismatch())
        hassector(codomain(t), f1.uncoupled) && hassector(domain(t), f2.uncoupled)
            # should an error thrown here
    end
```

```
    return sreshape(
            (StridedView(t.parent.data[c])[t.parent.rowr[c][f2],
t.parent.colr[c][f1]])',
            (dims(codomain(t), f1.uncoupled)..., dims(domain(t), f2.uncoupled)...))
end
@propagate_inbounds Base.setindex!(t::AdjointTensorMap{S, N₁, N₂}, v,
                        f1::FusionTree{I, N₁}, f2::FusionTree{I, N₂}) where {S,
N₁, N₂, I} =
    copy!(getindex(t, f1, f2), v)

@inline Base.getindex(t::AdjointTrivialTensorMap) =
    sreshape(StridedView(t.parent.data)', (dims(codomain(t))...,
dims(domain(t))...))
@inline Base.setindex!(t::AdjointTrivialTensorMap, v) = copy!(getindex(t), v)

@inline Base.getindex(t::AdjointTrivialTensorMap, ::Tuple{Nothing, Nothing}) =
getindex(t)
@inline Base.setindex!(t::AdjointTrivialTensorMap, v, ::Tuple{Nothing, Nothing}) =
    setindex!(t, v)

# For a tensor with trivial symmetry, allow direct indexing
@inline function Base.getindex(t::AdjointTrivialTensorMap, indices::Vararg{Int})
    data = t[]
    @boundscheck checkbounds(data, indices)
    @inbounds v = data[indices...]
    return v
end
@inline function Base.setindex!(t::AdjointTrivialTensorMap, v,
indices::Vararg{Int})
    data = t[]
    @boundscheck checkbounds(data, indices)
    @inbounds data[indices...] = v
    return v
end

# Show
#------
function Base.summary(t::AdjointTensorMap)
    print("AdjointTensorMap(", codomain(t), " ← ", domain(t), ")")
end
function Base.show(io::IO, t::AdjointTensorMap{S}) where {S<:IndexSpace}
    if get(io, :compact, false)
        print(io, "AdjointTensorMap(", codomain(t), " ← ", domain(t), ")")
        return
    end
    println(io, "AdjointTensorMap(", codomain(t), " ← ", domain(t), "):")
    if sectortype(S) == Trivial
        Base.print_array(io, t[])
        println(io)
    elseif FusionStyle(sectortype(S)) isa UniqueFusion
        for (f1, f2) in fusiontrees(t)
            println(io, "* Data for sector ", f1.uncoupled, " ← ", f2.uncoupled,
":")
            Base.print_array(io, t[f1, f2])
```

```
            println(io)
        end
    else
        for (f1, f2) in fusiontrees(t)
            println(io, "* Data for fusiontree ", f1, " ← ", f2, ":")
            Base.print_array(io, t[f1, f2])
            println(io)
        end
    end
end
```